Università degli studi di Pisa
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di laurea Specialstica in Informatica

Anno Accademico 2004-2005

Tesi di laurea

Metodo di Knowledge Discovery applicato ai Geographical Information Systems (GIS)

candidato

Luigi Scrimitore

Relatore:
Prof. Franco Turini

Controrelatore:
Prof. Paolo Mancarella

Correlatore:
Dott. Salvatore Rinzivillo

Non fa abbastanza un uomo onesto
se applica con esattezza e coscienza
l'arte che a lui è stata tramandata?
Se da giovane onori il padre tuo,
imparerai da lui volenteroso,
e se da uomo fatto accrescerai la scienza,
tuo figlio potrà giungere a mete ancor più alte.

Faust - Wolfgang Goethe

Ai miei genitori,
per la pazienza e
la fiducia dimostratami.

Indice

Indice delle tabelle e delle illustrazioni

Ringraziamenti

A tutti coloro che, col loro impegno e con la loro partecipazione diretta o indiretta, mi hanno aiutato nei momenti difficili.

Al prof. Turini, per la sua disponibilità e al dott. Rinzivillo, per la sua pazienza e lo spirito di dedizione dimostrato nel guidarmi passo passo in questa esperienza.

A tutti i miei amici “pisani”, a tutti coloro che pur non leggendo queste pagine hanno fatto parte della mia vita arricchendo il mio spirito con le loro esperienze, i loro sogni, le loro aspirazioni.

A Corneliu e ad Alba, che pur avendo preso strade diverse dalla mia, porterò per sempre con me nel cuore.

A Giuseppe, nelle cui profondità dell’animo mi sono specchiato.

A tutti i miei amici d’infanzia, che con pazienza e fiducia hanno atteso questo momento come fosse il loro.

A mio fratello, che con i suoi consigli puntuali ed il suo esempio, mi ha indicato il giusto cammino.

Ad Anna, che col suo Amore infinito mi ha reso più forte e fiducioso illuminando il mio presente ed il mio futuro.

Ai miei genitori, a mio padre e a mia madre, che mi hanno sorretto e guidato in ogni attimo del mio cammino, che mi hanno insegnato più di quanto possa apprendere in mille anni di studi.

A tutti costoro, non posso che dire **grazie**.

Luigi Scrimitore

Introduzione

La notevole crescita dei dati raccolti e memorizzati in databases ha creato la necessità di elaborarli al fine di ottenere conoscenze ed informazioni latenti e non facilmente rilevabili. Tale processo prende il nome di *data mining* o, più propriamente, *knowledge discovery in databases* (KDD).

I dati spaziali sono informazioni riguardanti oggetti che occupano uno spazio. Uno *spatial database* immagazzina, perciò, tali informazioni. Tra i vari modelli di rappresentazione dei dati spaziali, un cenno a parte meritano i *Geographic Information Systems* (GIS), in cui gli oggetti vengono rappresentati per mezzo di mappe.

I GIS e il processo di Knowledge Discovery in Databases (KDD) sono due tecnologie sviluppatesi con deboli correlazioni tra di loro. Recentemente, con l'accumulo di informazioni geografiche all'interno dei databases, si è realizzato l'effettivo potenziale delle informazioni contenute, rendendo logico, ed estremamente importante, l'uso delle tecniche di data mining applicate ai GIS.

L'esigenza di fondere due strumenti tecnologici così differenti infatti, nasce dalla consapevolezza che è lecito supporre che tra i dati aventi una valenza spaziale possono sussistere influenze e correlazioni tali da condizionare il comportamento e la distribuzione degli oggetti stessi presenti nel sistema. E' impensabile non ritenere che, ad esempio, la presenza di un mare nei pressi di una città non agisca su di essa e sulle dinamiche che la regolano.

Permettere di definire modelli in grado di prevedere andamenti, spiegare comportamenti all'apparenza complessi e di intervenire efficacemente sul territorio in esame è un aspetto essenziale in molti settori dell'economia, dell'industria, della ricerca e in generale in qualunque caso si debba analizzare dati geograficamente distribuiti.

Fino a pochi anni fa, gli unici strumenti utilizzati per risolvere tali problematiche erano basati su modelli statistico-descrittivi, spesso insufficienti o imperfetti nel rappresentare la realtà in quanto non in grado di rilevare le interdipendenze tra gli oggetti. Gli strumenti di analisi realizzati applicando tecniche di data mining ai dati presenti in un GIS, invece, sono in grado di evidenziare relazioni spaziali e patterns significativi non facilmente rilevabili tramite una semplice interpretazione visuale delle mappe.

In questa tesi, dopo aver evidenziato i metodi più noti in letteratura, si espone un sistema integrato in grado di combinare gli strumenti tradizionali di knowledge discovery con il modello di memorizzazione dei dati spaziali denominato GIS, al fine di ottenere patterns da cui ottenere "*regole assocative spaziali*" in grado di esprimere le correlazioni più importanti tra classi di oggetti spaziali.

Nel capitolo 1, si presentano alcuni tra i metodi più noti in letteratura di applicazione delle metodologie di data mining alle informazioni memorizzate nei GIS evidenziandone gli aspetti peculiari.

Nel capitolo 2, si espongono le nozioni di databases spaziale in generale e di GIS in particolare, specificando i metodi di accesso e di indicizzazione dello spazio rappresentato.

Nel capitolo 3, dopo aver introdotto per linee generali il processo di knowledge discovery, si definisce il concetto di regola associativa spaziale. In seguito, vengono mostrati due algoritmi tipici per la generazione di regole associative dato un insieme di transazioni: l'algoritmo Apriori e il FP-Tree (*Frequent Pattern Tree*).

Nel capitolo 4, si espone il metodo realizzato spiegando, tramite l'uso di un breve esempio, i vari passi di tale processo.

In particolare, nella prima parte, si spiega l'utilità nel definire gerarchie sia sulle relazioni topologiche espresse tramite lo standard del *9-intersection model* di Egenhofer, sia sugli oggetti presenti nel sistema informativo. Inoltre, si affronta la possibilità, fornita

all’utente, di estendere la gerarchia dei concetti con relazioni personalizzate e più complesse. Per quanto riguarda i dati, si definisce per ogni classe di oggetti, una struttura gerarchica al fine di permettere ricerche e analisi specialistiche su oggetti aventi caratteristiche peculiari.

Infine, dopo aver esposto il concetto di query spaziale tramite l’uso di uno psuedo linguaggio, si enunciano tutti i passi che dai dati grezzi presenti nel sistema conducono alla definizione delle regole associative spaziali, evidenziando per ognuno di essi le motivazioni e le strutture dati necessarie. In particolare, poiché le comuni tecniche per generare tali regole prendono in considerazione solo items semplici, si estende il metodo proposto al caso in cui si voglia effettuare analisi che prendano in considerazione l’eventualità che un items si possa ripetere.

Nel capitolo 5, si chiariscono i passi implementativi del metodo mettendo in risalto i momenti più importanti e complessi dell’applicazione.

Nel capitolo 6, si mostrano i risultati dell’applicazione realizzata ad un caso concreto.

Nelle conlusioni, si fa un sunto del problema affrontato e si accenna ai possibili futuri sviluppi da realizzare.

Capitolo 1
Stato dell’Arte

Fino a pochi anni fa, il mezzo più comune per analizzare dati spaziali è stato quello di utilizzare le analisi statistiche. Il problema più grande di questa tecnica però, risiede nell’assunzione, a priori, dell’indipendenza statistica dei dati spazialmente distribuiti. Questo vincolo è, spesso, non solo disatteso, ma anche irrealistico da proporre, in quanto, generalmente, ogni regione è influenzata dalle aree circostanti.

Successivamente, un approccio più ragionato ha applicato ai dati tecniche di generalizzazione (spaziali e non), al fine di renderli più ad alto livello possibile e applicando poi, con più facilità, tecniche di analisi sofisticate.

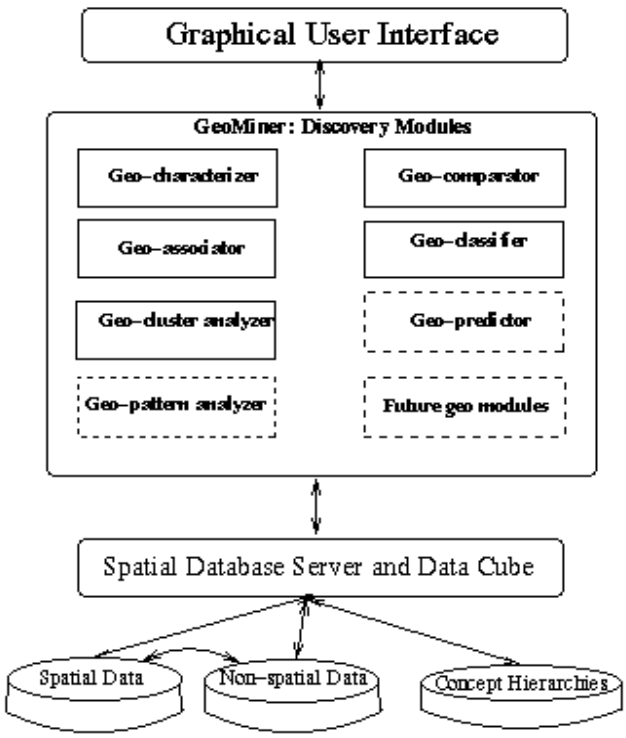
Le generalizzazioni che si possono definire, sono classificabili in due grandi aree concettuali:

- Nonspatial-dominant : si parte dai dati non spaziali. I dati numerici vengono raggruppati in intervalli, mentre quelli simbolici vengono descritti con valori più generali (es. mele, pere, arance, vengono tradotti in frutta). In questo modo, distinti valori di basso livello e selettivi, vengono accorpati in identici valori di alto livello. Ciò comporta che, aree geografiche diverse, vengano accorpate a formare un’unica zona (ad es. tutte le zone coltivate a mele, pere, arance, ecc. vengono raggruppate sotto l’indicazione di frutteti). In questo modo, si otterranno mappe con un basso numero di regioni aventi un alto livello descrittivo, utili a modellare situazioni quale ad esempio “*le case singole e costose a Vancouver sono raggruppate lungo la spiaggia e intorno a due parchi*”. [KH95]
- Spatial-dominant : prima si esegue un passo di generalizzazione sui dati spaziali (usando gerarchie fornite dall’utente o strutture dati gerarchiche), successivamente, si raggruppano i dati non-spaziali relativi alla stessa classe. Spesso però, le gerarchie non sono note a priori. In questo caso, è necessario descrivere il

comportamento spaziale di oggetti simili, o determinare le caratteristiche distintive di gruppi diversi.

Basandosi su quest'ultima tipologia di generalizzazione, Koperski e Han hanno implementato un metodo per determinare regole associative spaziali all'interno di un GIS denominato *GeoMiner*. **[**HKS98**]**

Tale tecnica si basa sul principio, sviluppato in qualunque processo di knowledge discovery, secondo cui in un vasto database possono esistere molte regole associative, di cui alcune, possono occorrere raramente, altre non essere sufficientemente interessanti. Quelle che appaiono con una frequenza relativamente alta (*largo supporto*) e con forti implicazioni (*alta confidenza*) sono definite *strong rules*. Esse esprimono le interconnessioni più importanti tra gli oggetti presenti nel database in questione.



Architettura di GeoMiner

Il processo di ricerca di spatial association rules in GeoMiner, può essere formalizzato in questo modo:

Dato in input:

- Un database spaziale su cui è definita una *gerarchia di concetti*,
- Una query con una classe di riferimento, un insieme di classi di oggetti spaziali rilevanti dette classi referenziate, e un insieme di relazioni spaziali su cui esiste una *gerarchia delle relazioni topologiche*,
- una gerarchia di *n* livelli di dettaglio L_i e, per ogni livello, una coppia di vincoli-soglia (*Min_Conf[i]* e *Min_Supp[i]*),

Trovare:

- tutte le regole associative spaziali forti e multi-livello tra gli oggetti della mia classe di riferimento e quelli delle classi rilevanti, utilizzando le relazioni definite in input.

Il metodo che essi hanno proposto è un processo costituito da progressivi raffinamenti e basato sulla classificazione sia delle relazioni che degli oggetti spaziali. In esso, si applica la query spaziale al database spaziale, selezionando i dati in base alla loro classificazione nella gerarchia di concetti definita. Tutti gli oggetti così trovati vengono inseriti in un nuovo database (*Task_Relevant_DB*).

Su questo nuovo DB, si applicano alcuni efficienti metodi di accesso spaziali (R-tree, K-B-Tree, Quadtree, ecc..), che restituiranno, per ogni singolo oggetto della classe di riferimento, un primo insieme di relazioni spaziali con gli oggetti delle classi referenziate, che andranno a confluire in un nuovo DB (*Coarse_predicate_DB*).

Per ogni predicato così determinato, si calcolerà il relativo supporto e lo si confronterà con il *Min_Supp[1]*, scartando tale relazione se non supera il vincolo minimo richiesto. In questo modo, si otterrà una prima, grezza, definizione di regole associative spaziali di livello 1. Combinando a coppie le relazioni così create, si otterranno predicati più complessi (di livello 2); tra questi, si sceglierano, poi, solo quelli aventi un supporto superiore alla soglia di secondo livello (*Min_Supp[2]*). Iterando questo processo finchè non

si ottiene un insieme di predicati di livello *n* vuoto, si genererà l'insieme di tutti i predicati di lunghezza qualsiasi.

Malerba, Esposito e Lisi hanno proposto una loro versione del metodo di Korpersky e Han, realizzando un sistema denominato SPADA [MEL01], in cui i dati e le loro relazioni sono visti come formule atomiche. In particolare, tale metodo permette di estrarre relazioni topologiche definite secondo la semantica del *9-intersection model* postulata da Egenhofer. L'algoritmo si basa sulla relazione d'ordine ≤ esistente tra 2 patterns definiti come congiunzione di atomi; in particolare, dati 2 patterns P_1 e P_2, se $P_1 \leq P_2$ allora *"P_1 è più generale di P_2"*, o *"P_2 è più specifico di P_1"* (quindi P_2 è ottenuto aggiungendo a P_1 altre informazioni). La generazione dei patterns, candidati alla creazioni di regole associative avviene iterando un processo di specializzazione in cui, i patterns determinati in un certo passo, vengono raffinati nel passo successivo sostituendoli con altri più specifici ottenuti aggiungendogli uno o più atomi. L'insieme dei candidati così ottenuti viene sottoposto ad una fase di *pruning*, in cui, i patterns non ottimali, determinati dal non superamento della condizione di minimo supporto consentito, vengono scartati ed estromessi dalle successive fasi di dettaglio.

Un altro metodo rilevante in letteratuta è quello di Ester, Frommelt, Kriegel e Sander [EFKS00]. In esso, il problema di fare knowledge discovery in un GIS, viene risolto definendo un *Neighborhood Graph* (grafo delle vicinanze), in cui si rappresentano tutte le possibili relazioni topologiche che un oggetto può avere con gli altri oggetti. In esso, ogni nodo è un oggetto del database e due nodi n_1 e n_2 sono connessi da un arco se e solo se vale la relazione generica *neighbor*(*object*(n_1), *object*(n_2)).

Il predicato *neighbor* può essere:

- una relazione topologica tra quelle definite nel 9-intersection model {*meet, overlap, covers, covered-by, contains, inside, equal*}.
- Una relazione metrica {*distance* < *d*}.
- Una relazione di posizionamento {*north, south, west, east*}.

Un *neighborhood path* è una listra di nodi [n_1, n_2, . . ., n_k] connessi da successivi archi, in cui cioè, il predicato *neighbor*(n_i, n_{i+1}) è verificato per ogni, $1 \leq i \leq k-1$ e *length* è il numero di nodi del path.

Su tale grafo, vengono implementate alcune operazioni base quali:

- *get_nGraph(db, rel)* che ritorna la porzione di grafo che è in relazione *rel* con l'oggetto *db*. *Rel* può essere anche una congiunzione dei predicati predefiniti.
- *get_neighborhood(graph, o, pred)* che ritorna l'insieme di tutti gli oggetti o_i che sono connessi da uno o più archi che soddisfano la condizione *pred* con *o*. *Pred* può essere un attributo spaziale o non-spaziale.
- *create_nPaths(objects, graph, pred, i)* genera tutti i path che partono da uno degli oggetti *objects* e avente *length* ≤ *i*.
- *extend(set_of_paths, graph, pred, i)* che ritorna l'insieme di tutti i paths di *set_of_paths* con più di *i* archi.

Infine, nell'esposizione del metodo, si enuncia come tali operazioni possono essere utilizzate in quattro differenti situazioni: *Spatial Association Rules*, *clustering*, *Spatial Trend Detection* e di *spatial classification*.

Capitolo 2
Gli Spatial Databases

Gli **Spatial DataBases** (**SDBs**) sono particolari basi di dati in cui le informazioni registrate, riguardano conoscenze inerenti oggetti geografici – es. terreni, strade, città, laghi, fiumi ecc. –. Essi vengono utilizzati per rappresentare un territorio in base alle sue caratteristiche.

L'applicazione di metodi e tecniche di data mining ai **SDBs**, viene utilizzata per permettere *l'estrazione di conoscenze implicite, di relazioni spaziali o di altri modelli non esplicitamente immagazzinati nei SDBs (Spatial Data Mining).* **[KAH95]**

Il nostro scopo è quello di estrarre correlazioni tra i dati dei nostri **SDBs** esprimibili con relazioni e connessioni non visibili, ad un primo e superficiale esame, tra valori di attributi relativi a classi di oggetti diverse.

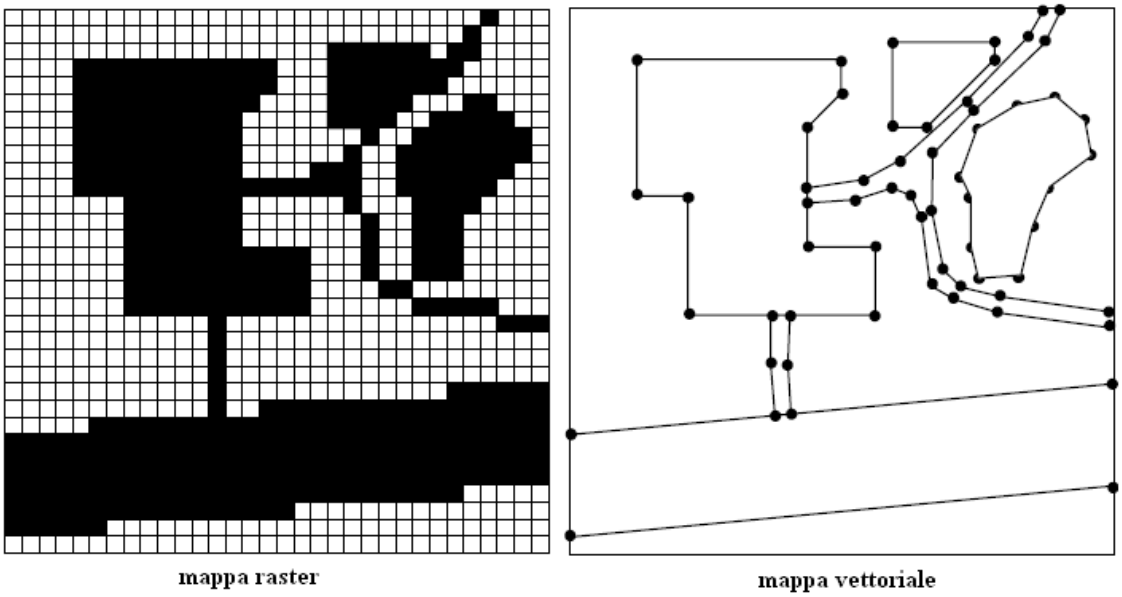
2.1 I GIS

Un geographical information system (GIS) è un genere speciale di sistema d'informazione, che permette di trattare, analizzare, sintetizzare, interrogare, pubblicare e prevedere i dati geograficamente relativi.

Quest'ultimi si compongono di:

- attributi spaziali (ad es. coordinate, geometria, forma) memorizzati in un SDB.
- attributi non-spaziali (ad es. nome della città, numero degli abitanti), inseriti in un DB relazionale.

I dati spaziali in un GIS vengono suddivisi e rappresentati tramite una gerarchia di livelli (**layers**), ognuno dei quali può essere rappresentato attraverso l'uso di una mappa **raster** o per mezzo di un modello **vettoriale**.



Rappresentazione raster e vettoriale di uno stesso layer

Il modello *raster* divide lo spazio in una griglia di celle, solitamente denominate *pixels*. Ogni cella contiene un singolo valore e la relativa posizione è definita dai relativi indici nella griglia. La risoluzione del modello raster dipende dal relativo formato del pixel. Più piccolo è il formato del pixel, più è alta la risoluzione, ma anche più grande è il formato di dati.

Il modello *vettoriale* rappresenta gli oggetti spaziali tramite l'uso di strutture dati, la cui primitiva di base è il punto. Ciò permette una rappresentazione più precisa delle coordinate ed è utile per effettuare analisi.

Le strutture dati principalmente utilizzate per memorizzare gli oggetti spaziali nel modello vettoriale, sono:

- *point,* definito tramite le sue coordinate.
- *line,* definita tramite i punti posti ai suoi vertici.

- *polygon*, sequenza di linee connesse in cui l'ultimo punto di una linea è anche il primo punto di un'altra linea. Inoltre le linee non devono intersecarsi.

2.2 Spatial data structures on GIS

Per rappresentare i dati spaziali presenti in un GIS in modo efficiente, è necessario utilizzare soluzioni ad-hoc, che ovviamente, influenzeranno notevolmente sia le prestazioni dell'intero sistema, che i risultati raccolti.

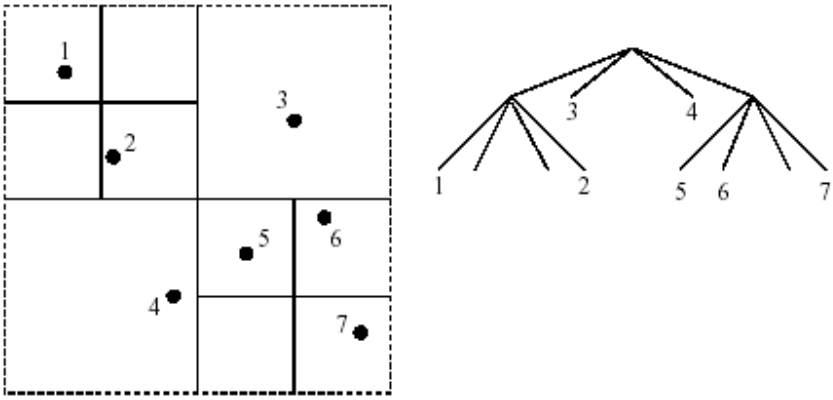
I metodi di accesso sono divisibili in due grandi aree:

- Point Access Methods **(PAMs)** in cui gli oggetti spaziali indicizzati sono dei punti (quelli non puntiformi vengono rappresentati prendendo il loro centro).
  Tra le tecniche che fanno parte di questa classe ricordiamo:
  K-Tree e derivati (k-d-Tree), (point) Quad-Tree e BSP-Tree.
- Spatial Access Methods **(SAMs)** in questo caso gli oggetti non sono puntiformi: Z-Ordering, R-Tree e varianti (R*/R+-tree, Hilbert R-tree, …) sono i metodi più noti. Essi si basano e usano il concetto di *minimal bounding box* (*MBB*) o *minimal bounding rectangle* (*MBR*) di un oggetto.

2.2.1 Quadtree

Il quadtree è usato per indicizzare lo spazio 2D. Ogni nodo interno dell'albero (ovvero ogni nodo non foglia), divide lo spazio in quattro sottospazi distinti chiamati NE, NO, SE e SO. Ognuno di questi sottospazi viene ulteriormente diviso, con un procedimento ricorsivo, finché non si verifica una di queste condizioni:

a) il sottospazio non contiene oggetti
b) il sottospazio contiene un solo oggetto
c) si è raggiunto il livello di dettaglio massimo.



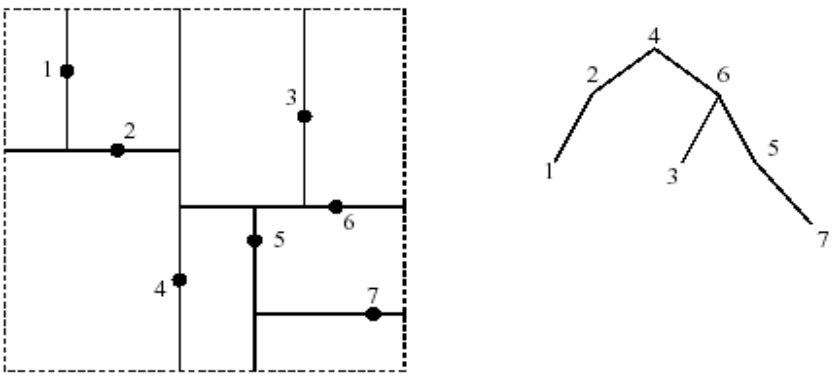
Quadtree

Il quadtree non è un albero bilanciato e il suo bilanciamento dipende dalla distribuzione dei dati nello spazio e dall'ordine di inserimento dei punti.

2.2.2 k–d–tree

È una variante dell'albero binario e suddivide lo spazio k-dimensionale su cui viene applicato.

Quest'albero suddivide lo spazio in due sottospazi distinti in accordo ad una delle due coordinate *x* e *y*.



K-d-tree

La fase di inserimento è simile a quella dell'albero binario.

Svantaggio: la struttura è sensibile all'ordine in cui gli oggetti vengono inseriti.

2.2.3 R–tree

È la variante del B-tree per i dati spaziali. Quest'albero è bilanciato e suddivide lo spazio in rettangoli che possono sovrapporsi.
Ogni nodo contiene da un minimo di m ed un massimo di M figli (dove $2 \leq m \leq M/2$). La radice contiene un minimo di 2 figli, tranne se è una foglia (in questo caso l'albero è vuoto).

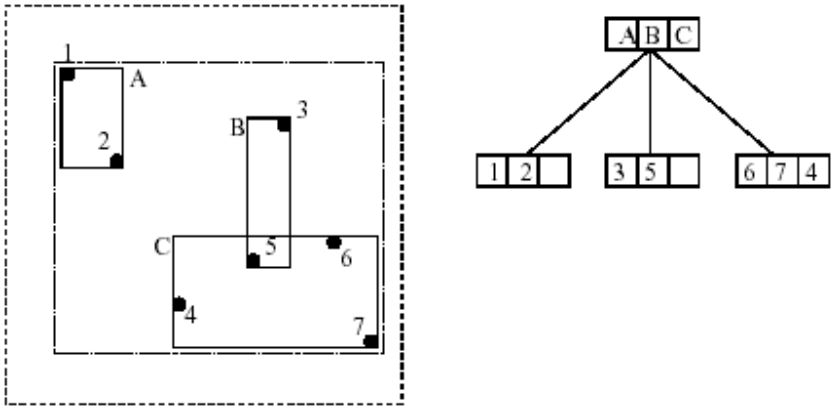
Ogni nodo è rappresentato tramite il *Minimum Bounding Rectangle* (*MBR*) che contiene tutti gli oggetti del relativo sotto-albero.
Ciascuno dei figli del nodo viene suddiviso ricorsivamente. I puntatori agli oggetti spaziali sono memorizzati nelle foglie.

A causa della sovrapposizione degli MBRs, potrebbe essere necessario cercare in più di un ramo dell'albero. Di conseguenza, è importante separare il più possibile i rettangoli. Questo problema deve essere risolto al momento dell'inserzione (funzione INSERT), utilizzando un'euristica adeguata.

Se si tenta di inserire un nuovo oggetto in un nodo foglia pieno, in cui cioè ho già memorizzato M nodi, questa operazione provoca la separazione della foglia con conseguente suddivisione degli oggetti tra le due nuove foglie. Questo processo può causare la spaccatura anche di nodi a livelli più alti, con conseguente risuddivisione degli oggetti. Anche la cancellazione può dare origine ad un'operazione di riassestamento dell'intero albero, se avviene in una foglia con solo m figli.

Poiché i MBRs possono sovrapporsi, l'operazione di ricerca può interessare più sottoalberi. È necessario perciò fare in modo di decrementare la probabilità di ricerche multiple.

L'R-tree è una delle strutture dati spaziale più citate ed è usato molto spesso come pietra di paragone per valutare la bontà di nuove tecniche.



R-tree o R*-tree

2.2.4 R–tree*

R*-tree è una variante dell'R–tree in cui si usano differenti euristiche per l'operazione di INSERT.

Mentre l'R–tree cerca di minimizzare l'area di tutti i nodi dell'albero, l'R*–tree combina più criteri insieme: l'area coperta da un bounding rectangle, il margine di un rettangolo e la sovrapposizione di più rettangoli.

L'obiettivo è quello di decrementare l'area coperta da un bounding rectangle riducendo così anche lo spazio in esso inutilizzato (*dead space*) e non presente in altri bounding rectangles. Questo decrementa il numero di rami che devono essere analizzati. La minimizzazione del margine, la somma cioè delle lunghezze dei lati di un bounding rectangle predilige l'uso di quadrati.

L'implementazione di questo metodo è più complessa, ma gli R*–trees sono prestazionalmente più potenti degli R-trees.

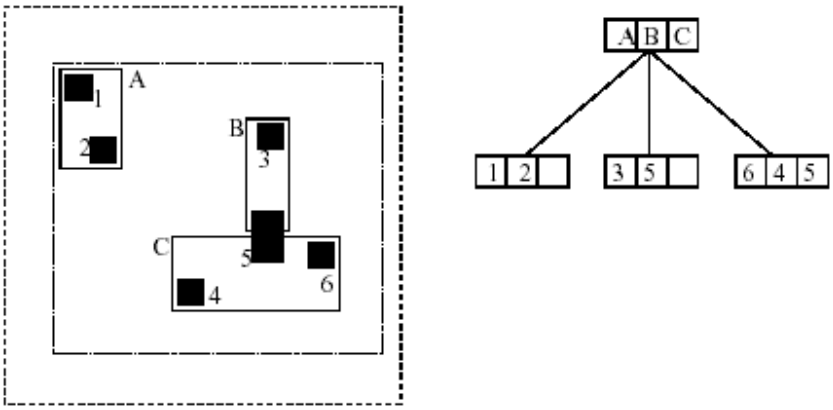
2.2.5 R+–tree

Il R+–tree è un'altra estensione del R–tree. In contrasto al R–tree, i bounding Rectangles dei nodi di uno stesso livello non si sovrappongono. Questa caratteristica decrementa il numero di rami in cui ricercare riducendo così anche il consumo di tempo.

Nel R+–tree è permesso dividere gli oggetti, cosicché, differenti sue parti possono essere memorizzate in differenti nodi di uno stesso livello.

Se un rettangolo si sovrappone ad un altro, lo decomponiamo in un gruppo di rettangoli non sovrapposti che coprono lo stesso oggetto.

Questo incrementa la dimensione dell'albero, con conseguente aumento dell'utilizzo dello spazio di memoria, ma annulla le sovrapposizioni tra nodi riducendo così il consumo di tempo al momento della ricerca.



R^+-tree

Capitolo 3
Knowledge Discovery in Databases e Data Mining

Oltre a estrapolare e interpretare i valori presenti in un database attraverso una comune interrogazione, potrebbe essere interessante a fini statistici e decisionali, rilevare relazioni apparentemente non evidenti tra insiemi di oggetti scorrelati.

Questo processo esplorativo prende il nome di *Knowledge Discovery in Databases* o brevemente *KDD*.

In letteratura esistono numerose definizioni di KDD, che differiscono principalmente per il settore in cui operano e per gli scopi per cui tale processo viene realizzato. Tra le tante possiamo ricordare:

"Data Mining, o Knowledge Discovery in Databases (Kdd) come è anche noto, è l'estrazione non banale di implicita, precedentemente sconosciuta, e potenzialmente utile informazione dai dati. Questo processo include un numero di differenti approcci, quali clustering, data summarizzation, apprendimento di classification rules, ricerca di reti di dipendenze, analisi dei cambiamneti, e ricerca di anomalie".[1]

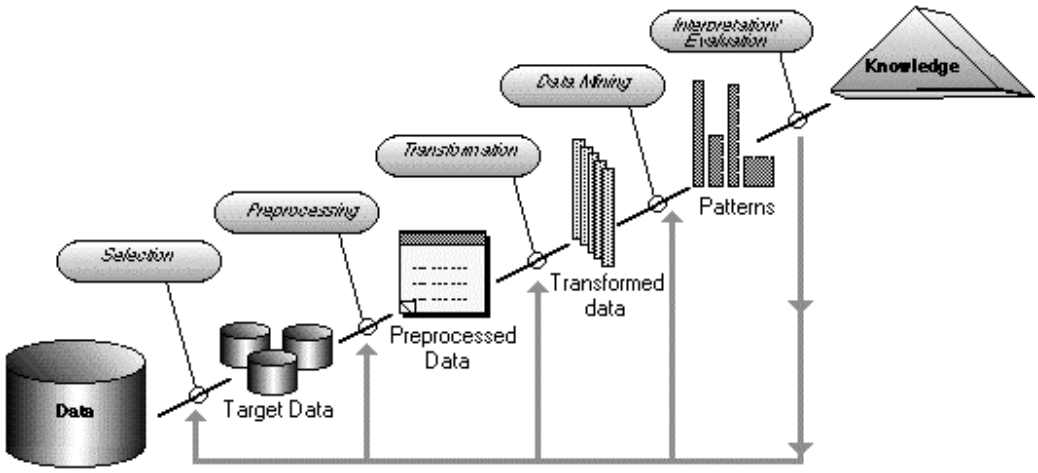
Il termine Data Mining è utilizzato come sinonimo di Knowledge Discovery in Databases (KDD), anche se sarebbe più preciso parlare di Knowledge Discovery quando ci si riferisce all'intero processo di estrazione della conoscenza, e di Data Mining come di una particolare fase del suddetto processo (la fase di applicazione di uno specifico algoritmo per l'individuazione dei "patterns").

[1] Cit. *William J Frawley, Gregory Piatetsky-Shapiro e Cristopher J Matheus*

Le fasi che portano dall'insieme grezzo dei dati all'estrazione della conoscenza possono essere riassunte in cinque punti:[2]

- **Selezione**: selezione o segmentazione dei dati secondo alcuni criteri.
- **Preprocessing**: ``pulizia" dei dati da certe informazioni ritenute inutili e che possono rallentare le future interrogazioni. In questa fase, inoltre, i dati possono essere trasformati per evitare eventuali inconsistenze dovute al fatto che dati simili possono provenire da sorgenti diverse e quindi con metadati leggermente diversi (ad esempio in un database il sesso di una persona può essere salvato come “m” o “f” ed in un altro come “0” o “l”).
- **Trasformazione**: i dati non sono semplicemente trasferiti da un archivio ad uno nuovo, ma sono trasformati in modo tale che sia possibile anche aggiungere informazione a questi, come per esempio informazioni demografiche comunemente usate nelle ricerche di mercato. Quindi i dati vengono resi “*usabili e navigabili*”.
- **Data Mining**: questo stadio si occupa di estrarre dei modelli dai dati. Un modello può essere definito come: dato un insieme di fatti *F* (i dati), un linguaggio *L* ed alcune misure di certezza *C*, un modello è una dichiarazione *S* nel linguaggio *L* che descrive le relazioni che esistono tra i dati di un sottoinsieme *G* di *F* con una certezza *c* tale che *S* sia più semplice in qualche modo della enumerazione dei fatti contenuti in *G*.
- **Interpretazione** e **valutazione**: i modelli identificati dal sistema vengono interpretati cosicché la conoscenza che se ne acquisisce può essere di supporto alle decisioni, quali ad esempio la previsione e la classificazione dei compiti, il riassunto dei contenuti di un database o la spiegazione dei fenomeni osservati.

[2] Cit. *Usama Fayyad & Evangelos Simoudis*



KDD process

3.1 Spatial Association Rules

Nel nostro caso, poiché i dati da analizzare hanno una forte caratterizzazione spaziale, il processo può a ragione essere specializzato in *spatial data mining*. Lo Spatial data mining è perciò, un sottocampo del data mining che tratta dell'*estrazione di conoscenza implicita, relazioni spaziali o altri interessanti modelli non esplicitamente memorizzati nei databases spaziali*.**[EKS97]**

Sia $I = \{i_1, i_2, \ldots, i_m\}$ un insieme di items. Sia D un database di transazioni dove ogni transazione T, è definita come un insieme di items tale che $T \subseteq I$. Sia A un insieme di items. Una transazione T contiene A se e solo se $A \subseteq T$.

Definizione:

Una *regola associativa* è una implicazione della forma $\boldsymbol{A \rightarrow B}$ **(s, c)**, in cui:

- ***A*** e ***B*** sono predicati costruiti come congiunzioni di items spaziali cioè: $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m)$.
- ***s*** è il **supporto** cioè la probabilità che i predicati *A* e *B* appaiano insieme nella stessa transazione $supporto(A, B) = Prob\{ A \cup B \}$.

- ***c*** è la **confidenza**, cioè la probabilità che B si verifichi se A si è verificato

$$\text{confidenza}(A \rightarrow B) = \text{Prob}\{B|A\} = \frac{\text{Prob}\{A \cup B\}}{\text{Prob}\{A\}} = \frac{\text{supporto}(A, B)}{\text{supporto}(A)}$$

Una regola associativa si defisce *spaziale* se gli items trattati sono proposizioni della forma <*relazione spaziale, oggetto*>.

Non tutte le possibili regole generabili sono utili. Per decidere se una regola associativa è valida dobbiamo comparare il supporto e la confidenza dei suoi predicati con due valori soglia detti **minimo supporto** ξ e **minima confidenza** γ forniti dall'utente.

Definizioni:

- Un insieme di predicati A si definisce *largo* se il suo supporto supera la soglia minima ($s \geq \xi$).
- Una regola $A \rightarrow B$ si definisce forte (*strong rule*) se il predicato composto $(A \wedge B)$ è largo e se la confidenza è alta ($c \geq \gamma$).

Un insieme di items viene detto più semplicemente *itemset*. Se esso contiene k items, viene definito *k-itemset*. Il numero di transazioni che contengono un itemset definisce la sua frequenza nel database. Un itemset soddisfa il minimo supporto se la sua frequenza supera o, almeno, eguaglia il prodotto tra il numero di transazioni totali e il minimo supporto ξ. Se un itemset soddisfa tale vincolo allora è detto **itemset frequente**. L'insieme di tutti i k-itemset frequenti è comunemente denotato con $\mathbf{L_k}$.

Proprietà:

- Un *itemset* si dice *frequente* se e solo se tutti i suoi items sono frequenti.
- Se un *k-itemset* ha frequenza *q*, qualunque *n-itemset* (con $n \geq k$), che lo contiene avrà frequenza $p \leq q$.
- Se un *k-itemset* è infrequente, qualunque *n-itemset* (con $n \geq k$), che lo contiene è infrequente (*proprietà di antimonotonia*).

Osservazione:

Se un itemset *I* non soddisfa il minimo supporto *s*, allora *I* non è frequente, quindi *Prob{I}* $< s$. Se un item *A* viene aggiunto all'itemset *I*, l'itemset ottenuto ($I \cup A$) non può occorrere più frequentemente di *I*, quindi *Prob{I* $\cup$ *A}* $<$ *Prob{I}* $< s$.

Perciò, tutti i sottoinsiemi non vuoti di un qualunque itemset frequente devono essere a loro volta frequenti.

Il processo di generazione delle regole associative (spaziali), può essere scomposto in due passi:

1. *Trovare tutti gli itemsets frequenti*.
2. *Generare strong association rules dagli itemsets frequenti*.

3.2 Algoritmi di data mining

Trovare tutti gli itemsets frequenti è un proccesso assai costoso, sia dal punto di vista del tempo impiegato che dallo spazio utilizzato.

3.2.1 Algoritmo Apriori

Il metodo più utilizzato per generare tutti gli itemsets frequenti è certamente l'Apriori. Esso si basa su un approccio *level-wise* in cui i *k-itemsets* vengono utilizzati per generare gli *(k+1)-itemsets*.

L'algoritmo affronta il problema scomponendolo in due fasi distinte:

1. *Join step*. Generazione dell'insieme di *k-itemsets candidati* $\boldsymbol{C_k}$ ottenuto fondendo insieme coppie di itemsets frequrenti del livello precedente $\mathbf{L_{k-1}}$ tali che unendoli essi abbiano (k-2) items in comune.

   $(\mathbf{L_{k-1}}$ *join* $\mathbf{L_{k-1}})$ = { A *join* B | A,B $\in \mathbf{L_{k-1}}$, |A . B| = k-2}

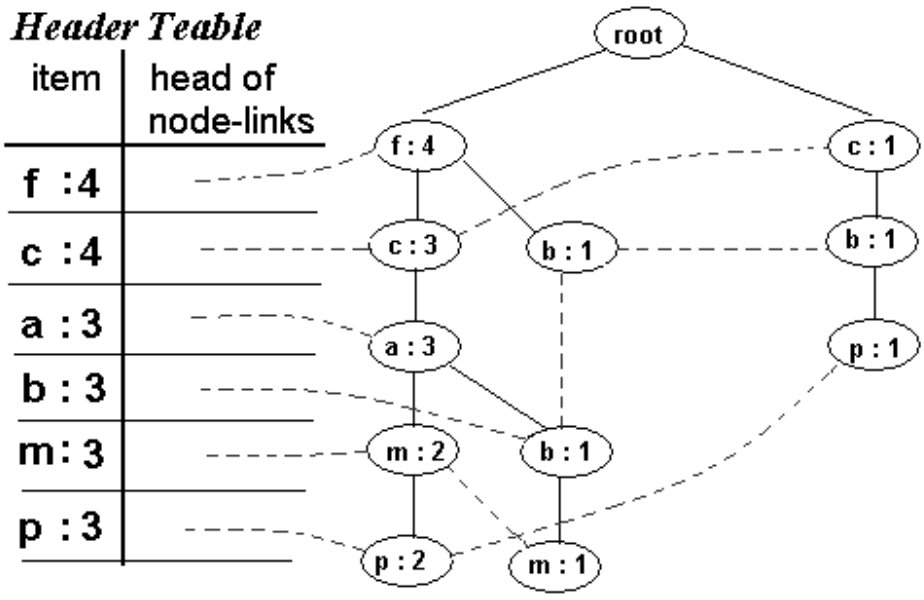
2. *prune step*. Selezione dall'insieme degli *k-itemsets candidati* $\boldsymbol{C_k}$, degli *k-itemsets frequenti*, confrontando la loro frequenza con il minimo supporto richiesto.

3.2.2 Il FP-Tree

Il FP-Tree o **Frequent Pattern Tree**, è uno strumento alternativo al comune algoritmo Apriori per generare tutti gli itemsets frequenti. **[HP04]**

In questo metodo, dopo una fase di decisione e una di ordinamento, si memorizzano tutte le transazioni presenti nel nostro database in modo compatto, attraverso l'uso di un albero: l'albero appunto, dei pattern frequenti.

Un FP-Tree è un albero avente, in ogni nodo un items e una frequenza. Ogni percorso generabile, cioè l'insieme dei nodi visitati partendo da un nodo generico per giungere alla radice, è un itemset, o meglio, un pattern canditato ad essere frequente. La frequenza del pattern sarà data dal valore presente nel nodo di altezza minima.



esempio di un FP-Tree

Poiché, come da proprietà, un itemset è frequente se e solo se tutti gli items di cui è composto sono a loro volta frequenti, per generare gli itemsets frequenti, possiamo non tenere conto degli items infrequenti. Per questo motivo, si visita tutto il database transazionale contando il numero di volte che ogni item appare.

TID	Items presenti
1	ACDFMP
2	ABCEFM
3	BGF
4	BCP
5	ACHFMP

Database delle transazioni

A	B	C	D	E	F	G	H	M	P
3	3	4	1	1	4	1	1	3	3

Tabella degli item e relativa frequenza

Per ogni items, si controlla se verifica il vincolo di minimo supporto: in questo modo divideremo l'insieme degli items di partenza in due gruppi, quelli frequenti (utili per le nostre analisi) e quelli che appaiono un numero limitato di volte nelle transazioni (che sono da scartare).

F	C	A	B	M	P	D	E	G	H
4	4	3	3	3	3	1	1	1	1

Tabella degli items ordinati per supporto decrescente

Rivisitando tutto il database, per ogni transazione si scarteranno gli items che sono infrequenti e si ordineranno i restanti, per frequenza decrescente.

TID	*Items esistenti*	*Items frequenti ordinati*
1	**a,c,**d,**f,m,p**	**f,c,a,m,p**
2	**a,b,c,**e,**f,m**	**f,c,a,b,m**
3	b,**g,**f	f,b
4	**b,c,p**	c,b,p
5	a,c,**h,**f,m,p	f,c,a,m,p

Database delle transazioni con gli items ordinati

Arrivati a questo punto, i dati sono ottimizzati per essere inseriti nel FP-Tree.

Partendo dalla prima transazione, ordinata e ripulita, creiamo un nuovo nodo per ogni item presente nella transazione, ottenendo così un primo percorso. Ogni nuovo nodo inserito avrà come frequenza il valore 1. Passiamo alla seconda transazione, prendendo il primo item F, notiamo che nell'albero costruito esiste un percorso con F. Quindi incrementiamo la frequenza del nodo F e scendiamo nell'albero; lo stesso discorso vale per il secondo e il terzo items (C e A), mentre per il quarto (B), non troviamo nessun percorso, quindi aggiungiamo al nodo a cui siamo giunti finora il nuovo figlo di B. Iterando il processo fino al termine del database otterremo il FP-Tree in figura.

La visita dell'albero, alla ricerca dei patterns frequenti, richiede una struttura dati di supporto, l'*header table*, per permettere il facile raggiungimento di tutti i nodi etichettati con un dato item.

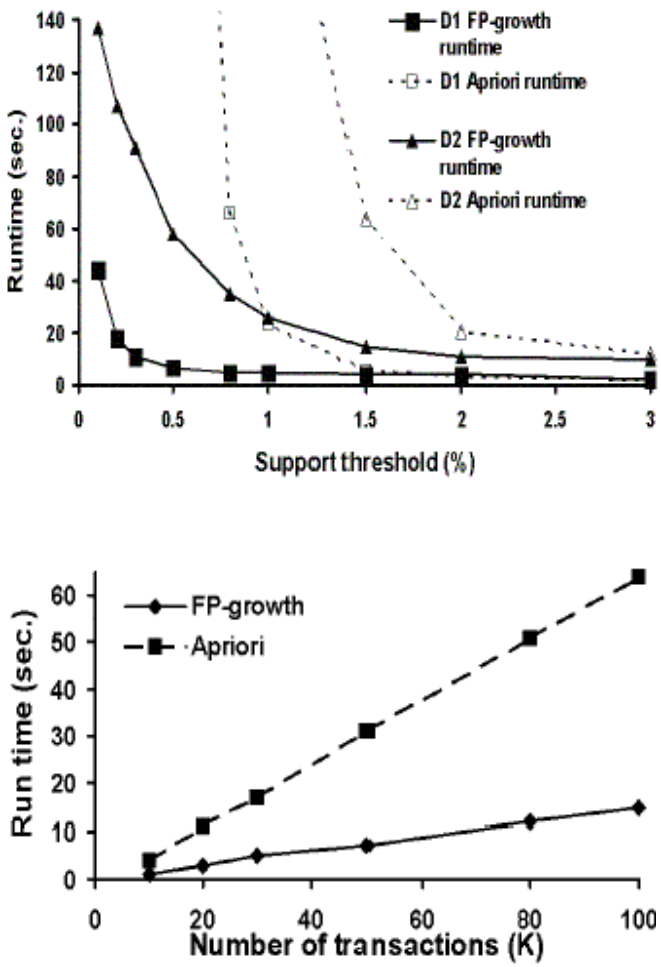
Partendo dai nodi più in basso e risalendo l'albero si genererà un insieme di patterns candidati; tra questi si scarteranno tutti quelli aventi frequenza, che corrisponde alla frequenza del nodo più in profondità, inferiore al minimo supporto.

3.2.3 Apriori vs FP-Tree

In questo paragrafo, confrontiamo le prestazioni di FP-Growth, l'algoritmo di ricerca su FP-Tree, con il classico algoritmo Apriori.

Se il dataset è sparso e il minimo supporto è alto, gli itemsets frequenti sono brevi e sono pochi; in questa situazione il vantaggio di FP-Growth con Apriori non è così evidente. Non appena si abbassa la soglia del supporto, la differenza diventa più evidente. Il vantaggio di FP-Growth sull'Apriori diviene importante quando il dataset contiene un miscuglio di transazioni brevi e lunghe. La differenza diviene incolmabile poi, se il dataset ha patterns lunghi, in quanto Apriori è costretto a generare un notevole numero di candidati.

In definitiva, FP-Growth presenta notevoli vantaggi quando la soglia del supporto è bassa e quando il numero di transazioni è ampio e possiamo affermare che, FP-Growth è un ordine di grandezza più veloce dell'Apriori.



Grafici di confronto tra Apriori e FP-Growth sulla scalabilità

Capitolo 4
Esempio di KDD su GIS

In questo capitolo, esporremo il metodo di analisi dei dati e di generazione delle regole associative spaziali proposto, avvalendoci di una query spaziale d'esempio ed evidenziando ogni singola fase di tale processo.

Si affronteranno, inoltre, le problematiche relative alla rilevazione della conoscenza aggiuntiva derivante dalla ripetizione frequente di coppie <*relazione*,*classe*> all'interno del nostro sistema.

4.1 La gerarchia dei concetti

Dopo aver costruito le basi di dati spaziali e le relative strutture dati necessarie per indicizzarle e prima di avviare il processo di ricerca, dobbiamo definire i predicati utili per le *spatial association rules* ed il loro significato.

I predicati o relazioni spaziali esprimono le relazioni topologiche esistenti tra gli oggetti. La loro definizione è uno dei momenti fondamentali nel processo di spazial data mining, in quanto esse, più di qualunque altra informazione, influiscono sulla determinazione degli itemsets e, conseguentemente, sulla specificazione delle regole spaziali.

4.1.1 Il 9-intersection model di Egenhofer

Il 9-intersection model è il più popolare strumento matematico per formalizzare le relazioni topologiche esistenti tra due oggetti spaziali.

In questo modello, si dà una formale definizione delle relazioni geometriche che possono esistere tra due arbitrari oggetti geometrici A e B, confrontando le 9 possibili intersezioni che si possono avere tra l'interno, i confini e l'esterno dei due oggetti.

Di ogni oggetto *A* interessa conoscere:

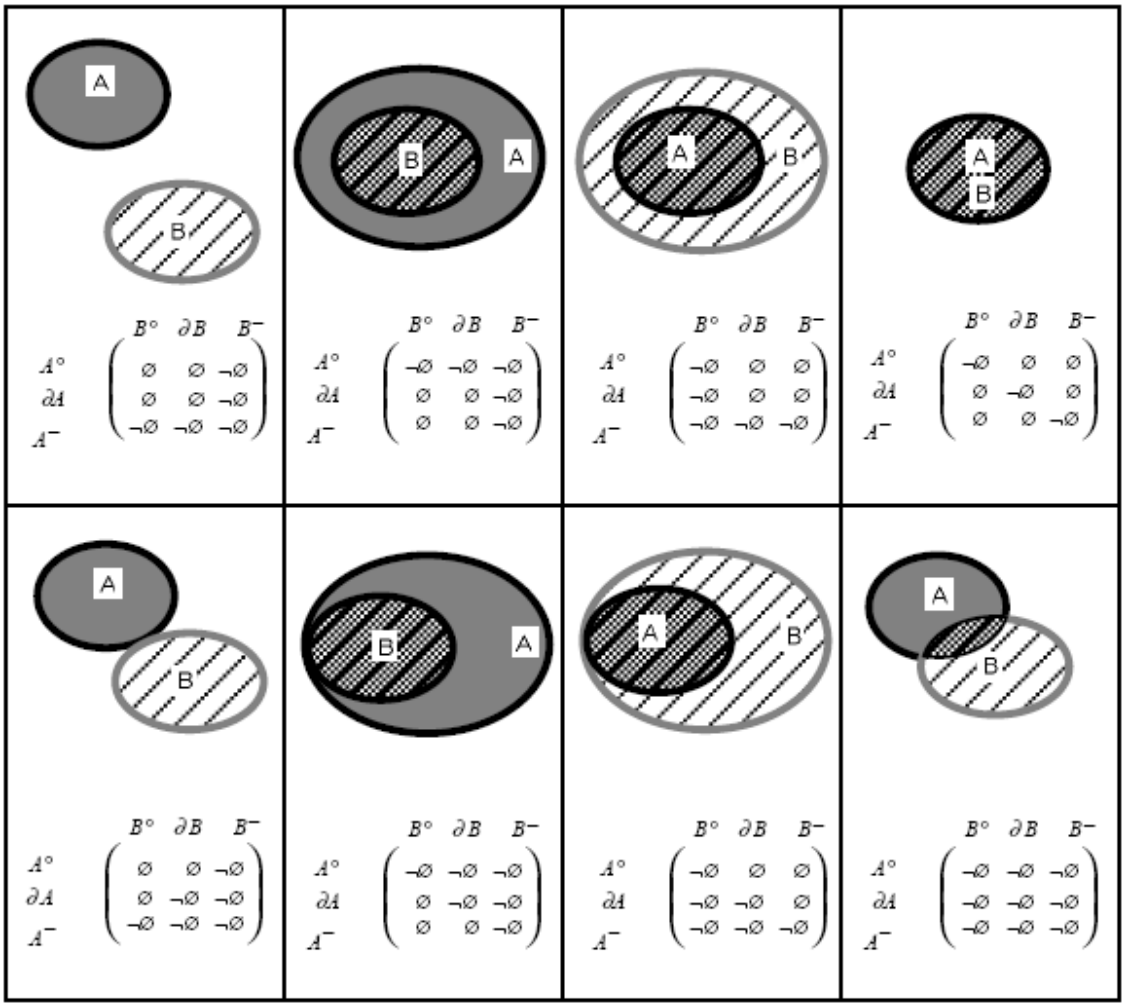
- Il *confine* ∂A ovvero il bordo della figura *A*.
- L'*interno* A° la porzione di spazio racchiusa dal confine di *A*.
- L'*esterno* A^- definito come il complemento di *A*, ovvero tutto l'universo tranne la porzione di spazio racchiusa da *A*.

La possibile relazione esistente tra due oggetti *A* e *B* viene definita, calcolando le nove intersezioni tra il confine, l'interno e l'esterno dell'oggetto *A* con il confine, l'interno e l'esterno di *B*. Queste nove intersezioni vengono raccolte in una matrice di dimensione 3x3 contenente, in ogni suo elemento, l'informazione vuoto/non-vuoto [Ø, ¬Ø].

$$R_9(A,B) = \begin{bmatrix} \partial A \cap \partial B & \partial A \cap B^\circ & \partial A \cap B^- \\ A^\circ \cap \partial B & A^\circ \cap B^\circ & A^\circ \cap B^- \\ A^- \cap \partial B & A^- \cap B^\circ & A^- \cap B^- \end{bmatrix}$$

Matrice delle interezioni

Nel 9-intersection model, considerando i diversi valori (vuoto o non vuoto), che le nove intersezioni possono avere, è possibile definire un insieme di relazioni topologiche. **[EF91]** Per esempio, otto relazioni diverse possono essere identificate tra due regioni spaziali in R^2: *disjoint*, *meet*, *equal*, *inside*, *contains*, *covers*, *covered-by* e *overlap*. **[ES93]**



Interpretazione geometrica delle 8 possibili relazioni tra due regioni A e B.

Ad esempio, se volessimo esprimere, mediante il modello 9-intersections basato sul valore "*vuoto/non-vuoto*" delle intersezioni, la configurazione in cui la regione A "*covers*" la regione B avremo:

$$R(A,B) = \begin{pmatrix} A^\circ \cap B^\circ = \neg\emptyset & A^\circ \cap \partial B = \emptyset & A^\circ \cap B^- = \emptyset \\ \partial A \cap B^\circ = \neg\emptyset & \partial A \cap \partial B = \neg\emptyset & \partial A \cap B^- = \emptyset \\ A^- \cap B^\circ = \neg\emptyset & A^- \cap \partial B = \neg\emptyset & A^- \cap B^- = \neg\emptyset \end{pmatrix}$$

o più brevemente:

$$R(A,B) = \begin{pmatrix} \neg\emptyset & \emptyset & \emptyset \\ \neg\emptyset & \neg\emptyset & \emptyset \\ \neg\emptyset & \neg\emptyset & \neg\emptyset \end{pmatrix}$$

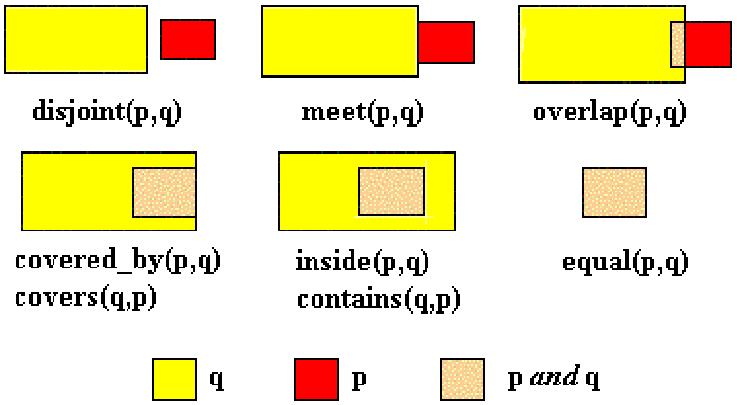
Similmente al caso *area-area*, in cui cioè entrambi gli oggetti sono delle regioni, si possono definire le relazioni topologiche esistenti anche nei casi:

- *area-linea*,
- *area-punto*,
- *linea-linea*,
- *linea-punto* e
- *punto-punto*.

4.1.2 Le relazioni spaziali e la gerarchia dei concetti

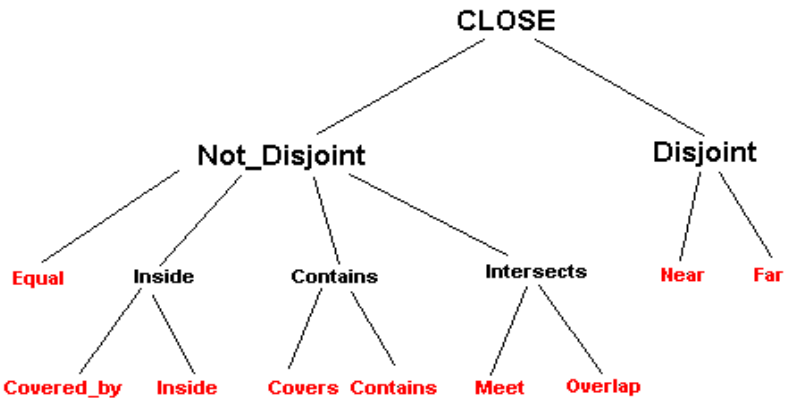
Basandoci su questo modello, i più importati predicati spaziali che possiamo utilizzare per esprimere le relazioni tra due oggetti *p* e *q* sono:

- **Disjoint(*p*,*q*)**. I Due oggetti sono separati.
- **Equal(*p*,*q*)**. I due oggetti hanno la stessa forma e dimensione e occupano lo stesso spazio.
- **Covered_by(*p*,*q*)** L'oggetto *p* è coperto dall'oggetto *q*.
- **Inside(*p*,*q*)** *p* è contenuto strettamente in *q*.
- **Covers(*p*,*q*)** *p* ricopre *q*.
- **Contains(*p*,*q*)** L'oggetto *p* contiene al suo interno l'oggetto *q*.
- **Meet(*p*,*q*)** I due oggetti si toccano.
- **Overlap(*p*,*q*)** I due oggetti si sovrappongono.



Possibili relazioni topologiche tra due oggetti

Poiché i predicati sopra descritti non sono scorrelati tra di loro ma anzi, spesso modellano situazioni simili (ad esempio le relazioni spaziali *inside* e *covered_by* configurano due circostanze analoghe), può essere utile, al fine di permettere analisi con relazioni meno specializzate, costruire una gerarchia sui predicati. Tale struttura è detta anche **gerarchia dei concetti**.



Gerarchia dei concetti

4.1.3 Inserimento di nuove relazioni

L'utente che interroga il GIS, potrebbe essere interessato, ai fini delle sue analisi, a conoscere relazioni, che intercorrono tra gli oggetti del mio sistema, più complesse di quelle previste nel 9-intersection model di Egenhofer.

Una qualunque relazione spaziale che un utente può creare viene definita:

- *Aggiuntiva* se viene utilizzata affiancando le precedenti.
- *Sostitutiva* se si sostituisce a relazioni esistenti. Quest'ultimo caso comporta una ristrutturazione dell'albero gerarchico dei concetti.

Poiché, però, le relazioni sopradefinite sono le minime possibili, qualunque nuova relazione spaziale può essere rappresentata come composizione di queste. Questa fusione tra relazioni viene realizzata ridefinendo la gerarchia dei concetti (definita sopra), per permettere al momento dell'interrogazione del R+-tree (o dello strumento utilizzato per indicizzare un layer), di ricavare le nuove relazioni.

Poniamo ad esempio, che all'utente non interessi una distinzione tra *Covered_by(p,q)* e *Inside(p,q)* e tra *Covers(p,q)* e *Contains(p,q)* e voglia trattarli come casi appartenenti alla stessa relazione. Definiremo allora due nuovi predicati:

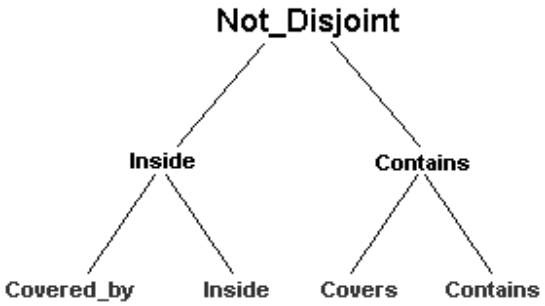
- ***Include(p,q)*** se l'oggetto *p* è incluso nell'oggetto *q* (cioè l'area occupata dall'oggetto *p* è contenuta in quella dell'oggetto *q*)
- ***included_by(p,q)*** se l'oggetto *p* include *l'oggetto q* (viceversa)

Questi due nuovi predicati, si possono esprimere come composizione di relazioni semplici tramite una semplice formula logica:

Include(p,q) = *(Covers(p,q))* ***OR*** *(Contains(p,q))*

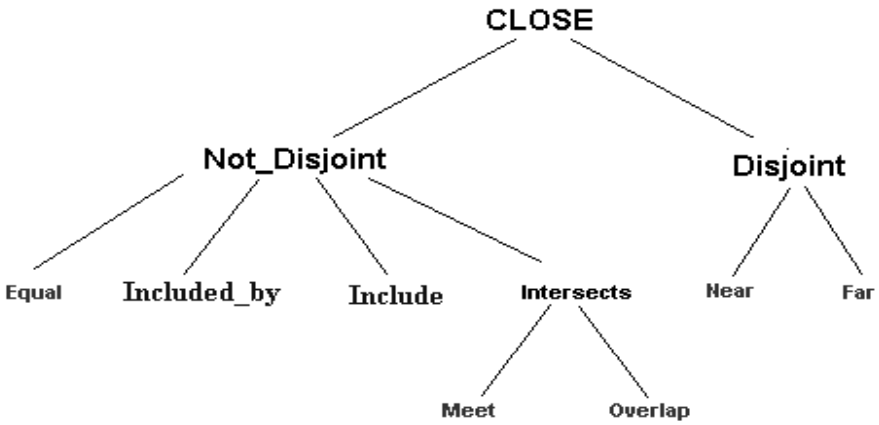
Included_by(p,q) = *(Covered_by(p,q))* ***OR*** *(Inside(p,q))*

Tale sostituzione, coinvolgerà la gerarchia dei concetti predefinita, in quanto la porzione dell'albero interessata dalla trasformazione sarà sostituita dalla rappresentazione dei due predicati logici.



sottogerarchia interessata

La nuova struttura gerarchia diverrà:



Nuova gerarchia dei concetti

In generale, possiamo affermare che qualunque nuova relazione definibile da un utente può essere espressa tramite una formula logica, in cui gli atomi sono certamente alcune delle relazioni definite nel 9-intersection model.

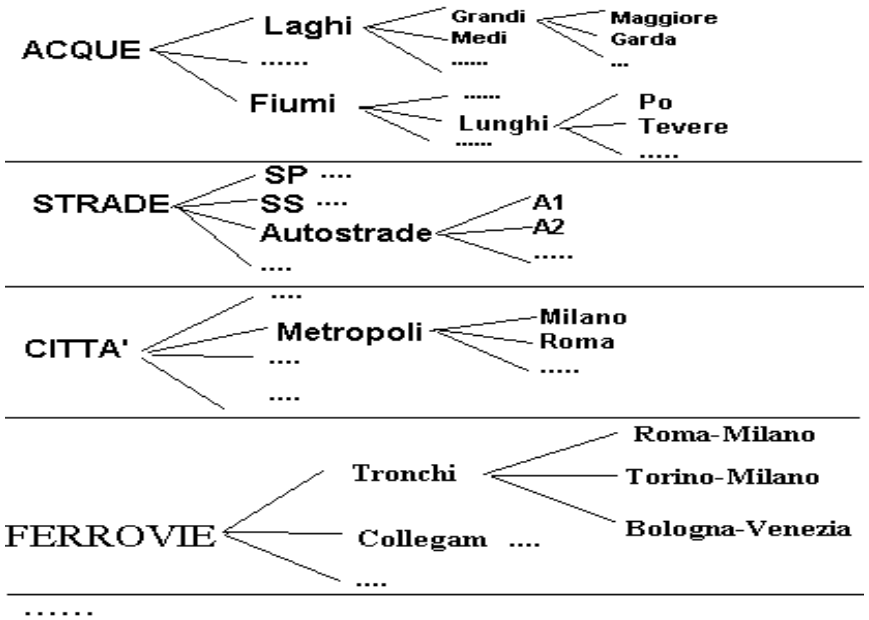
4.2 Le gerarchie sugli oggetti

Tutti gli oggetti appartenenti ad una stessa classe (es. le città), godono di caratteristiche specializzanti (es. la grandezza, la popolazione). Sfruttando tali attributi, possiamo pilotare il processo di analisi al fine di ottenere risultati che siano più interessanti per gli scopi per il

quale è stato messo in pratica. L'utente, infatti, potrebbe essere interessato a regole associative che non si riferiscono ad un'intera classe ma a sottoporzioni di essa, ovvero ai suoi sottoinsiemi aventi una determinata caratteristica in comune.
Ad esempio, potrebbe essere interessato a strade che abbiano un grosso volume di traffico, piuttosto che a conoscere informazioni relative all'intera classe delle *strade*, la quale include anche piccole strade di comunicazione quali "*strade provinciali*" e "*strade comunali*".

Per questo motivo, è auspicabile strutturare ogni singola classe di oggetti (e conseguentemente ogni singolo layer), in sottoclassi via via più specializzate, in modo tale da rendere il processo di selezione degli oggetti più naturale e variegato possibile.

Ad esempio, immaginiamo di analizzare i dati geografici relativi agli oggetti presenti nei seguenti layers: **Città**, **Strade, Bacini Idrici, Ferrovie** e **Altre Strutture**.
All'interno di ogni singola classe, possiamo classificare gli oggetti in base al valore da loro assunto per uno specifico attributo; iterando questo processo fino a quando non si è raggiunto un livello di specializzazione soddisfacente, si genererà, per ogni classe di oggetti, una gerarchia simile a quella in figura.



Gerarchie sugli oggetti

Nel nostro esempio, possiamo immaginare di suddividere gli oggetti che sono *Bacini Idrici* in base alla loro tipologia in *Fiumi*, *Laghi* e *Mari*. In seguito, possiamo ulteriormente spezzare ogni singola sottoclasse così ottenuta in base alla grandezza degli oggetti, ottenendo così:

- laghi grandi, medi e piccoli,
- fiumi lunghi o brevi,
- ecc.

Al termine di questo processo di specializzazione, nelle foglie della gerarchia così creata, si troveranno gli oggetti veri e propri.

Il problema principale di questo approccio è che tale gerarchia è statica, cioè una volta creata può essere analizzata soltanto seguendo le linee guida che ci hanno spinto nella sua definizione. Questo fatto impedisce, o rende difficoltoso, il riuso di una tale struttura, soprattutto nei casi in cui si desidera effettuare analisi per evidenziare conoscenze implicite slegate dalla gerarchica fornita. Un utente, ad esempio, potrebbe essere interessato a conoscere informazioni relative ai flussi di traffico indipendentemente dalla tipologia della strada. In questo caso, la gerarchia appena definita sulla classe *Strade*, è non solo inutile ma anche deleteria.

E' auspicabile, per questa ragione, invece di utilizzare la *gerarchia implicita* fornita dal sistema, qualora l'utente ne fosse interessato, permettere la specifica, tramite semplice formalismo, di una *gerarchia esplicita* in cui, i diversi valori dell'attributo *p-esimo*, definiscono i differenti valori presenti nel p-esimo livello di specializzazione della gerarchia.

CLASSE($attributo_1$, **…** , $attributo_k$)

Es. *Bacini Idrici*(Tipo, Dimensione, Navigabilità, …).

Come abbiamo precedentemente accennato, un utente può essere interessato a conoscere le relazioni che esistono tra gli oggetti della mia classe referente e i sottoinsiemi definiti nella gerachia della classe referenziata; ogni sottoinsieme può essere definito come

l'insieme di tutti gli oggetti aventi, per uno specifico attributo un valore ben preciso o quanto meno appartenente ad un range di valori predefinito.

Per le nostre analisi, potremmo non essere interessati a tutti i sottoinsiemi definiti, ma solo a quelli aventi determinate caratteristiche (ad es. potremmo suddividere le città in base ai loro abitanti e scegliere solo quelle aventi tale numero compreso tra 10.000 e 100.000). In questo caso, oltre a specificare l'elenco ordinato degli attributi interessanti per definire la gerarchia esplicita, possiamo specificare per alcuni di essi (o per tutti), il sottoinsieme dei valori significativi.

Classe(**attributo 1**(*valore1, …, valore n*), … , **attributo n**(*valore1, …, valore m*))

Es. *Bacini Idrici*(Tipo("*Fiume", "Lago"*), Dimensione (), Navigabilità (*"SI"*), …).

4.3 La query spaziale

La query spaziale è lo strumento attraverso cui l'utente esterno interroga il GIS, al fine di ottenere informazioni statistico-descrittive sul modello rappresentato.

Nella query, l'utente deve specificare almeno questi dati:

- La classe referente,
- L'insieme delle classi referenziate,
- Gli eventuali vincoli da porre sugli oggetti (per poter selezionare solo quelli utili),
- Il minimo supporto e la minima confidenza.

Un esempio di query spaziale potrebbe essere:

"Seleziona tutte le regole associative spaziali esistenti tra le città con almeno 500.000 abitanti con: i fiumi e i laghi, le strade a grande percorrenza, le ferrovie, i poli industriali, le zone agricole.

Inoltre, queste regole associative spaziali devono avere un supporto e una confidenza rispettivamente del 60% e del 80%".

Una sua rappresentazione in pseudo-linguaggio potrebbe essere:

Select *Spatial Association Rules*

From C *Città*;

To S *Strade*, **A** *Bacini_Idrici*, **Fe** *Ferrovie*, **M** *Monti*, **St** *Strutture;*

Where (C.popolazione ≥ 500.000**),**

((A.tipo = *"Fiume"***) or (A.tipo =** *"Lago"***) or (A.tipo = "***Mare"***),**

((S.tipo = *"Autoatrada"***) or (S.tipo = "***Super Strada***")),**

(M.altezza ≥ 1000**);**

Support 60%;

Confidence 80%;

4.4 Le fasi dell'analisi dei dati

Per eseguire correttamente la nostra query spaziale sugli oggetti appartenenti ai layers, è necessario passare attraverso una sequenza di fasi, ognuna delle quali è specializzata nella raccolta, organizzazione o strutturazione dei dati.

- *Fase 1: Raccolta dei dati*. In questa fase, per ogni oggetto della classe referente (le cittá), si visitano gli R+-trees utilizzati per indicizzare lo spazio dei layers del GIS (uno per ogni layer) e si raccolgono le relazioni, definite nella gerarchia dei concetti, esistenti tra gli oggetti della mia classe referente e quelli presenti nelle foglie dell'albero visitato.
- *Fase 2: Creazione del databases delle transazioni*. Si astraggono le coppie <*relazione, oggetto referenziato*> individuate nella fase precedente, sostituendo l'oggetto con la classe (o se specificato un maggior livello di dettaglio, la sottoclasse) di appartenenza.
- *Fase 3: Calcolo delle frequenze*. Per ogni transazione presente nella tabella creata al passo precedente, si verifica (e si memorizza), la presenza o meno di ogni possibile coppia <*relazione, classe*> definibile.

- *Fase 4: Raccolta degli items frequenti*. Si rileva, visitando la tabella delle frequenze creata al passo precedente, quali siano le coppie frequenti e quali no, confrontando il loro supporto con il *minimo supporto* ξ.
- *Fase 5: Generazione degli itemsets frequenti.* Utilizzando un algoritmo appropriato si generano a partire dagli items frequenti, tutti i possibili itemsets di lunghezza qualunque.
- *Fase 6: Generazione delle regole associative spaziali*. Si analizzano i k-predicati (o i k-itemsets) frequenti e si generano tutte le possibili regole associative spaziali confrontandole poi con il vincolo di *minima confidenza*.

Fase 1: Visita degli R+-tree e rilevazione delle relazioni

Per rappresentare efficacemente il nostro esempio, supponiamo di indicizzare lo spazio di ogni singolo layer con un R+-Tree. L'uso di un R+-Tree è consigliabile in quanto, pur essendo più costoso, in fase di creazione, di altri metodi alternativi, permette di spezzare i *minimal bounding rectangle* (*MBR*) che delimitano gli oggetti in più parti, permettendo uno spreco dello spazio notevolmente minore, soprattutto in quei casi in cui un oggetto attraversa in diagonale gran parte dello spazio (nel nostro esempio, lo spazio da indicizzare è l'intera penisola italiana; l' *MBR* di una autostrada come la A1, che da Napoli raggiunge Milano, occupa gran parte dello spazio totale, rendendo vana l'indicizzazione).

Ogni oggetto della classe referente, che abbia superato gli eventuali vincoli imposti, viene confrontato con gli oggetti indicizzati negli R+-trees definiti. Questo processo avviene percorrendo l'albero dalla radice verso le foglie, confrontando l'*MBR* dell'oggetto referente con quello relativo ai figli del nodo in cui mi trovo e scegliendo di visitare solo quelli con cui si ha una intersezione. Terminata la visita e giunti alle foglie, si procede con la rilevazione delle relazioni, definite nella gerarchia dei concetti, esistenti tra l'oggetto geometrico vero e proprio e ogni singolo oggetto presente nella foglia.

Qualora fosse necessario, si può effettuare, in questa fase, un ulteriore passo di raffinamento tramite un successivo livello di SAMs o PAMs.

Le coppie <*relazione, oggetto referenziato*> individuate, verranno inserite in una tabella avente una riga per ogni oggetto referente e una colonna per ogni layer.

CITTÀ	STRADE	ACQUE	FERROVIE	MONTI	STRUTTURE
Firenze	{<Meet,**A1**>, <Meet,**A11**>, <Overlap,**SS2**>, <Overlap,**SS222**>, <Overlap,**SS66**>, <Overlap,**SS62**>, <Overlap,**SS67**>, <Overlap,**SS302**>}	{<Overlap,**Arno**>}	{<Overlap,**RM-MI**>, <Overlap,**FI-PI**>}	{<Meet, **Appenennino Toscano**>}	{<Contain,**Staz.SMN**>, <Contain,**StazCmpDMarte**>, <Contain, **Monumenti**>, <Meet,**Aree Agricole**>, <Meet,**Zona Industriale**>}
Milano	{<Meet,**A1**>, <Meet,**A4**>, <Meet,**A8**>, <Overlap,**SS412**>, <Overlap,**SS11**>, <Overlap,**SS233**>, <Overlap,**SS33**>}	{<Overlap, **Lambro** >}	{<Overlap,**RM-MI**>, <Overlap,**MI-TO**>}	{}	{<Covered_by,**Tangenz**>, <Contain,**Staz.MilanoC**.>, <Contain,**Staz2**.>, <Contain,**Metropolitana**>, <Meet,**Zona Industriale**>}
Roma	{<Meet,**A24**>, <Meet,**A12**>, <Overlap,**SS6**>, <Overlap,**SS7**>, <Overlap,**SS8**>, <Overlap,**SS3**>, <Overlap,**SS4**>, <Overlap,**SS493**>}	{<Overlap, **Tevere**>}	{<Overlap,**RM-MI**>, <Overlap,**RM-NA**>}	{}	{<CoveredBy,**Racc.Anulare**> <Contain,**Staz**.**Termini**>, <Contain,**Staz.Ostiense**>, <Contain,**Staz.S.Pietro**>, <Contain,**Staz.Trastevere**>, <Contain,**Metropolitana**>, <Contain,**Monumenti**>, <Meet, **ZonaIndustriale**>}
Napoli	{<Meet,**A1**>, <Meet,**A3**>, <Meet,**SS162**>, <Overlap,**SS268**>, <Overlap,**SS7**>, <Overlap,**SS18**>}	{<Meet, **MarTirreno**>}	{<Overlap,**RM-NA**>, <Overlap,**NA-RC**>}	{<Meet,Vesuvio>}	{<Contain, **Porto**>, <Meet, **Zona Industriale**>, <Overlap,**Transvesuviana**>}
Palermo	{<Meet,**A20**>, <Meet,**SSxx**>, <Overlap,**SS624**>}	{<Meet, **MarTirreno**>}	{<Overlap,**PA-ME**>}	{}	{< Contain,**Porto**>, <Contain,**Staz**>, <Meet, **Aree Agricole**>}
Torino	{<Meet,**A4**>, <Meet,**A5**>, <Meet,**A8**>, <Meet,**A21**>, <Meet,**A32**>, <Meet,**SS64**>, <Meet,**SS10**>, <Meet,**SS11**>, <Meet,**SS393**>, <Meet,**SS460**>, <Overlap,**SS20**>}	{<Overlap,**F.Dora**>, <Meet, **Po**>}	{<Overlap,**MI-TO**>, <Overlap,**BO-TO**>}	{<Meet, Alpi>}	{<Overlap, **Metropolitana**>, <Contain,**Staz PortaNuova**>, <Contain,**Staz 2**>, <Meet, **Zona Industriale**>}
Bologna	{<Meet,**A1**>, <Meet,**A14**>, <Meet,**A13**>, <Meet,**SS64**>,	{<Meet, **F.Reno**>}	{<Overlap,**LE-MI**>, <Overlap,**RM-MI**>, <Overlap,**BO-TO**>}	{}	{<Overlap, **Zona Industriale**>, <CoveredBy, **Tangenziale**>, <Contain,**StazBologna C**>, }

	<Meet,**SS568**>, <Overlap,**SS65**>, <Overlap,**SS9**>, <Overlap,**SS569**>, <Overlap,**SS253**>}				
Genova	{<Meet,**A10**>, <Meet,**A12**>, <Meet,**SS1**>, <Overlap,**SS45**>}	{<Meet, **MarTirreno**>, <Overlap,**Trebbia**>	{<Overlap,**RM-FR>,** <Overlap,**GE-TO>**}	{}	{<Contain, **Porto**>, <Contain**,Staz Principe**>, <Meet, **Zona Industriale**>}

Tab. 1: relazioni oggetto-oggetto

Fase 2: Creazione della tabella delle relazioni inter-layer

Poiché le *regole associative spaziali* esprimono le informazioni implicite esistenti tra le classi del sistema in esame e non, invece, tra gli oggetti in esse contenute, astraiamo le coppie <*relazione*, *oggetto_referenziato*>, individuate al momento della visita degli indici spaziali (e memorizzate nella tabella 1), ottenendo per ogni città referente l'insieme delle relazioni che essa possiede con ogni singola classe.

Inoltre, poiché le regole che relazionano le classi possono non essere sufficientemente significative per i nostri scopi, possiamo sfruttare le gerarchie definite sugli oggetti e determinare così regole più specializzanti, ad esempio, potrebbero essere piú utili regole che relazionino grandi città con fiumi, laghi o mari piuttosto che quelle che mettono in riferimento tali città coi bacini idrici in generale.

Per questo motivo, spezziamo la classe *Bacini Idrici* in due sottoclassi "*Mari"* e *"Fiumi e Laghi*" e la classe *Strade* in "*Autostrade*" e "*SuperStrade*".

Ovviamente, le sottoclassi così definite non devono avere oggetti in comune.

Per semplicità, d'ora in poi, rappresenteremo le nuove coppie <*relazione, classe*> per mezzo di codici semplificativi, creati fondendo insieme l'abbreviazione della relazione seguita da quella della classe.

<Contains, Strutture>	**CnS**	<Meet, Strutture>	**MS**
<Covered_by, Strutture>	**CbS**	<Meet, SuperStrade>	**MSS**
<Meet, Autostrade>	**MA**	<Overlap, Ferrovie>	**OF**
<Meet, Fiumi e Laghi>	**MFL**	<Overlap, Fiumi e Laghi>	**OFL**
<Meet,Mari>	**MM**	<Overlap, SuperStrade>	**OSS**

Abbreviazioni usate

	Relazioni
FI	MA MA OSS OSS OSS OSS OSS OFL OF OF CnS CnS CnS MS MS
MI	MA MA MA OSS OSS OSS OSS OFL OF OF CbS CnS CnS CnS MS
RM	MA MA OSS OSS OSS OSS OSS OSS OFL OF OF CbS CnS CnS CnS CnS CnS CnS MS
NA	MA MA MSS OSS OSS OSS MM OF OF CnS MS OS
PA	MA MSS OSS MM OF CnS CnS MS
TO	MA MA MA MA MA MSS MSS MSS MSS MSS OSS OFL MFL OF OF OS CnS CnS MS
BO	MA MA MA MSS MSS OSS OSS OSS OSS MFL OF OF OF OS CbS CnS
GE	MA MA MSS OSS OFL MM OF OF CnS CnS MS

Tabella1.1: relazioni *oggetto-classe* abbreviate

Fase 3: Calcolo delle frequenze e creazione della tabella delle frequenze

Una volta ottenuti tutti i dati e averli raccolti nella tabella 1.1 delle relazioni *oggetto-classe*, è necessario elaborarli in base alla loro frequenza, passando attraverso varie fasi di raffinamento, affinchè sia possibile utilizzarli in maniera efficiente.

Per contare ed in seguito valutare la frequenza di ogni coppia <*relazione*, *classe*>, bisogna creare una nuova tabella avente una riga per ogni oggetto referente e una colonna per ogni possibile coppia <*relazione*, *classe*> generabile dal prodotto cartesiano dell'insieme delle relazioni con l'isieme delle classi (o sottoclassi) coinvolte nell'analisi.

Se ad esempio, visitando la tabella 1.1, nella transazione *i* vi occorre la coppia *j*, inseriremo nella cella (*i*, *j*) della tabella 2 un 1. Viceversa, se una coppia *k* non occorre nella transazione *i* la corrispondente cella (*i*, *k*) avrà valore 0.

CITTÀ	**AUTOSTRADE**							**SUPER STRADE**						
	E	Cb	Cs	I	Cn	**M**	O	E	Cb	Cs	I	Cn	**M**	**O**
Firenze						**1**								**1**
Milano						**1**								**1**
Roma						**1**								**1**
Napoli						**1**							**1**	**1**
Palermo						**1**							**1**	**1**
Torino						**1**							**1**	**1**
Bologna						**1**							**1**	**1**
Genova						**1**							**1**	**1**
frequenza	0	0	0	0	0	**8**	0	0	0	0	0	0	**5**	**8**

CITTÀ	**FIUMI e LAGHI**							**MARI**						
	E	Cb	Cs	I	Cn	M	**O**	E	Cb	Cs	I	Cn	M	O
Firenze							**1**							
Milano							**1**							
Roma							**1**							
Napoli													1	
Palermo													1	
Torino							**1**							
Bologna							**1**							
Genova							**1**						1	
frequenza	0	0	0	0	0	0	**6**	0	0	0	0	0	3	0

CITTÀ	STRUTTURE							FERROVIE						
	E	Cb	Cs	I	**Cn**	**M**	O	E	Cb	Cs	I	Cn	M	**O**
Firenze					**1**	**1**								**1**
Milano		1			**1**	**1**								**1**
Roma		1			**1**	**1**								**1**
Napoli					**1**	**1**	1							**1**
Palermo					**1**	**1**								**1**
Torino					**1**	**1**	1							**1**
Bologna		1			**1**		1							**1**
Genova					**1**	**1**								**1**
frequenza	0	3	0	0	**8**	**7**	3	0	0	0	0	0	0	**8**

Tabelle delle presenze degli items nelle transazioni

Fase 4: Raccolta degli items frequenti

Poiché, ai fini della nostra analisi, siamo interessati solo a predicati frequenti, tra tutti i possibili dobbiamo segliere quelli la cui frequenza è maggiore della soglia di minimo supporto ($\xi = 60\%$), richiesta in input con la query spaziale.

Calcolando la sommatoria di tutti i valori presenti su una colonna della tabella 2, è possibile determinare il numero di transazioni in cui la relativa coppia appare; nel nostro caso, il numero di città che ha quella precisa relazione con la classe indicata.

Scartando quelle aventi un basso supporto, abbiamo definito l'insieme delle coppie frequenti ovvero l'insieme degli *items frequenti* e, di conseguenza, anche l'insieme $\mathbf{L_1}$ degli itemsets di lunghezza 1.

Nome relazione	**MA**	**OSS**	**CnS**	**OF**	**MS**	**OFL**	**MSS**
Nuovo codice	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Frequenza	**8**	**8**	**8**	**8**	**7**	**6**	**5**

Tabella degli items frequenti

Al fine di rendere più leggibili i successivi risultati, semplifichiamo i codici usati finora, sostituendoli con una lettera progressiva dell'alfabeto.

Fase 5: Generazione degli itemsets frequenti

La generazione degli itemsets frequenti può essere svolta applicando uno degli algoritmi classici di data mining. Per chiarezza e per confronto, esponiamo il processo di generazione degli itemsets frequenti sia con l'algoritmo Apriori che utilizzando il FP-Tree.

Fase 5.1: Applicazione dell'algoritmo Apriori

Una volta definiti quali items sono frequenti, applicando l'algoritmo Apriori, possiamo generare successivi insiemi di candidati $\mathbf{C_k}$ di lunghezza via via crescente; valutando, poi, la frequenza degli itemsets in $\mathbf{C_k}$ otterremo l'insieme $\mathbf{L_k}$ degli itemsets frequenti lunghi k, con *k* qualsiasi.

{A,B}, {A,C}, {A,D}, {A,E},
{A,F}, {A,G},
{B,C}, {B,D}, {B,E}, {B,F},
{B,G},
{C,D}, {C,E}, {C,F}, {C,G},
{D,E}, {D,F}, {D,G},
{E,F}, {E,G},
{F,G}

C_2: Insieme dei candidati di dimensione 2

AB	8	**BC**	8	**CD**	8	**DE**	7	**EF**	5
AC	8	**BD**	8	**CE**	7	**DF**	6		
AD	8	**BE**	7	**CF**	6	**DG**	5		
AE	7	**BF**	6	**CG**	5				
AF	6	**BG**	5						
AG	5								

L_2: Insieme degli itemsets di dimensione 2

{A,B,C}, {A,B,D}, {A,B,E},
{A,B,F}, {A,B,G},
{A,C,D}, {A,C,E}, {A,C,F},
{A,C,G},
{A,D,E}, {A,D,F}, {A,D,G},
{A,E,F},
{B,C,D}, {B,C,E}, {B,C,F},
{B,C,G},
{B,D,E}, {B,D,F}, {B,D,G},
{B,E,F},
{C,D,E}, {C,D,F}, {C,D,G},
{C,E,F},
{D,E,F}

C_3: Insieme dei candidati di dimensione 3

ABC	**8**	**ACD**	**8**	**BCD**	**7**	**CDE**	**7**	**DEF**	**5**
ABD	**8**	**ACE**	**7**	**BCE**	**7**	**CDF**	**6**		
ABE	**7**	**ACF**	**6**	**BCF**	**6**	**CDG**	**4**		
ABF	**6**	**ACG**	**5**	**BCG**	**5**	**CEF**	**5**		
ABG	**5**	**ADE**	**7**	**BDE**	**7**				
		AEF	**5**	**BDF**	**6**				
				BDG	**5**				
				BEF	**5**				

L_3: Insieme degli itemsets di dimensione 3

{A,B,C,D}, {A,B,C,E}, {A,B,C,F}, {A,B,C,G},
{A,B,D,E}, {A,B,D,F}, {A,B,D,G}, {A,B,E,F}, {A,B,E,G}, {A,B,F,G},
{A,C,D,E}, {A,C,D,F}, {A,C,D,G}, {A,C,E,F}, {A,C,E,G},{A,C,F,G},
{B,C,D,E}, {B,C,D,F}, {B,C,D,G}, {B,C,E,F}, {B,C,E,G}, {B,C,F,G},
{B,D,E,F}, {B,D,E,G}, {B,D,F,G},
{C,D,E,F}, {C,D,E,G}, {C,D,F,G}

C_4: Insieme dei candidati di dimensione 4

ABCD	8	**ABDE**	7	**ACDE**	7	**BCDE**	7	**BDEF**	5	**CDEF**	5
ABCE	7	**ABDF**	6	ACDF	6	**BCDF**	6				
ABCF	6	**ABDG**	5	**ACDG**	5	**BCDG**	5				
ABCG	5	**ABEF**	5	**ACEF**	5	**BCEF**	5				

L_4: Insieme degli itemsets di dimensione 4

ABCDE	7	**ABDEF**	5	**ACDEF**	5	**BCDEF**	5	**CDEF**	5
ABCDF	6								
ABCDG	5								
ABCEF	5								

L_5: Insieme degli itemsets di dimensione 5

ABCDEF	5

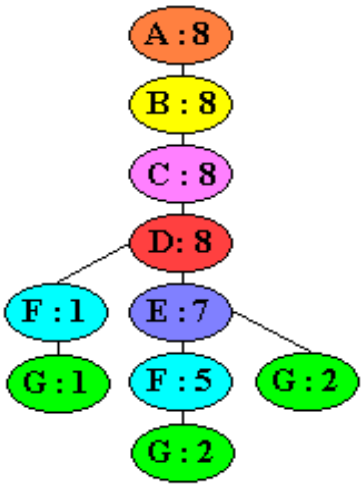
L_6: Insieme degli itemsets di dimensione 6

Fase 5.2 Utilizzo di un FP-Tree

Per applicare questo metodo, è necessario ordinare (per frequenza decrescente) gli items presenti in ogni transazione e scartare da quest'ultima, gli items che non hanno superato il vincolo di minimo supporto richiesto.

	Relazioni	**Rel. ordinate**
FI	**A B F D C E**	A B C D E F
MI	**A B F D C E**	A B C D E F
RM	**A B F D C E**	A B C D E F
NA	**A B D C E G**	A B C D E G
PA	**A G B D C E**	A B C D E G
TO	**A G B F D C E**	A B C D E F G
BO	**A G B D C F**	A B C D F G
GE	**A G B F D C E**	A B C D E F G

Tabella delle relazioni ordinate



Creazione di un FP-Tree senza ripetizioni

Essendo il FP-Growth (il metodo per determinare i k-pattern basato sul FP-Tree) e l'Apriori due algoritmi equivalenti, i risultati generati saranno i k-itemsets generati precedentemente e con la stessa frequenza.

Fase 6: Generazione delle regole associative spaziali

Una volta generati tutti gli itemsets frequenti, il processo di determinazione delle regole associative spaziali esistenti consiste in una semplice definizione di tutte le possibili regole ottenute combinando in ogni modo possibile gli items di uno stesso itemsets e sottoponendo quest'ultimi a un test di validità, quale quello della minima confidenza richiesta.

4.5 Generazione di itemsets con ripetizioni

Spesso, le coppie di relazioni che si individuano, appaiono in molte transazioni ripetute più volte. Questo fatto esprime connessioni inter-layers più complesse di quelle che abbiamo espresso finora, considerando solo le coppie non ripetute. Ad esempio, molte città sono in relazione con più superstrade.

Per questo motivo potremmo pensare a regole associative più specifiche, tali da evidenziare maggiormente questo legame.

Per considerare questo fatto, è necessario modificare in parte il processo visto finora, agendo su alcune fasi dell'analisi, pur senza modificarne il procedimento generale.

Dopo aver visitato gli R+-Trees e aver rilevato le coppie <*relazione, classe*> (fase 1 e 2 del processo di analisi), dobbiamo calcolare le loro frequenze (fase 3). Qui, non ci limitiamo solo a verificare la presenza o meno di un predicato in una transazione, ma contiamo anche il numero di volte che tale predicato appare.

La tab. 2 non sarà, perciò, una tabella di bit di verità (assenza/presenza), ma ogni sua cella conterrà un valore intero.

CITTÀ	AUTOSTRADE							SUPER STRADE						
	E	Cb	Cs	I	Cn	M	O	E	Cb	Cs	I	Cn	M	O
Firenze						2								6
Milano						3								4
Roma						2								6
Napoli						2							1	3
Palermo						1							1	1
Torino						5							5	1
Bologna						3							2	4
Genova						2							1	1

CITTÀ	FIUMI e LAGHI							MARI						
	E	Cb	Cs	I	Cn	M	O	E	Cb	Cs	I	Cn	M	O
Firenze							1							
Milano							1							
Roma							1							
Napoli													1	
Palermo													1	
Torino							2							
Bologna							1							
Genova							1						1	

CITTÀ	STRUTTURE							FERROVIE						
	E	Cb	Cs	I	Cn	M	O	E	Cb	Cs	I	Cn	M	O
Firenze					3	2								2
Milano		1			3	1								2
Roma		1			6	1								2
Napoli					1	1	1							2
Palermo					2	1								1
Torino					2	1	1							2
Bologna		1			1		1							3
Genova					2	1								2

Tabelle riassuntive delle frequenze in ogni transazione degli items

Questa visualizzazione mette in evidenza la presenza di relazioni identiche tra un oggetto della classe referente (le città) e gli oggetti di una classe referenziata (es. le Autostrade). A vista, è possibile notare, infatti, che è comune incontrare una città avente la stessa relazione spaziale con oggetti diversi appartenenti alla stessa classe. Ad esempio "*Firenze meet due autostrade, overlap sei SuperStrade, ecc.*".

Oltre alle coppie frequenti, perciò, dobbiamo verificare se vi siano delle ripetizioni di queste, anch'esse frequenti. Quindi per ogni items, dobbiamo verificare anche quante sue ripetizioni sono frequenti e scartare tutte quelle che eccedono tale numero.

Ad esempio, il predicato <*meet, Autostrade*> è presente in tutte e 8 le transazioni; la stessa relazione ripetuta 2 volte è presente in 7 delle 8 totali (supporto del 87.5%), mentre la stessa relazione ripetuta per tre volte si ha solo in 3 transazioni (37.5%). Quindi, tale coppia è da considerarsi frequente, così pure lo è se replicata due volte, mentre la presenza contemporanea di tre <*meet, Autoatrade*> in una stessa transazione è infrequente e, quindi, questo caso non dovrà essere preso in considerazione.

In aggiunta al procedimento con copppie non ripetute, creeremo una nuova tabella in cui si avrà una riga per ogni colonna movimentata della tabella 2 (cioè una riga per ogni tipo di coppia presente nella tabella 1.1) e *n* colonne in cui memorizzare la frequenza delle 2,3,…,*n* ripetizioni.

COD	RELAZIONI	1	2	3	4
MA	**<Meet, Autostrade>**	**8**	**7**	3	
MSS	**<Meet, SuperStrade>**	**5**	3		
OSS	**<Overlap, SuperStrade>**	**8**	**5**	**5**	4
OFL	**<Overlap, Fiumi eLaghi>**	**6**	4		
	<Meet, Mari>	3			
	<Covered_by, Strutture>	3			
CnS	**<Contain, Strutture>**	**8**	**6**	3	
MS	**<Meet, Strutture>**	**7**	1		
	<Overlap, Strutture>	3			
OF	**<Overlap, Ferrovie>**	**8**	**7**	1	

Tab. 3: frequenze delle relazioni usate e delle loro ripetizioni

Visitando questa tabella e confrontando i valori contenuti in ogni sua cella con il minimo supporto (ξ = 60%), possiamo notare che solo 7 predicati sono frequenti (su 42 possibili). Contando poi le loro ripetizioni il numero sale a 12.

Per semplicità di notazione, d'ora in poi ogni relazione frequente (evidenziata in verde nella tab. 3), verrà identificata semplicemente da una abbreviazione. Inoltre, se la relazione ripetuta *k* volte è anch'essa frequente, si creerà una nuova relazione avente la stessa sigla del predicato unario seguita dall'indice *k* (es. *MA2*, *OSS2*, *OSS3*). Facendo questa distinzione, sarà più agevole trattare in maniera indipendente, come se fossero predicati differenti, le coppie e le loro ripetizioni frequenti.

Dopo aver individuato i predicati frequenti, compresi anche quelli ripetuti, creiamo una nuova tabella avente una struttura simile alla tabella 2 (una riga per ogni transazione ma con una colonna per ogni predicato frequente e non per ogni possibile predicato riscontrabile), in cui le colonne sono ordinate per frequenza decrescente. In questa tabella, se il predicato *j* apparirà nella transazione *i*, la cella (*i*,*j*) avrà valore 1, altrimenti il suo valore sarà zero.

CITTÀ	MA	OSS	CnS	OF	MS	MA 2	OF 2	CnS 2	OFL	MSS	OSS 2	OSS 3
Firenze	1	1	1	1	1	1	1	1	1		1	1
Milano	1	1	1	1	1	1	1	1	1		1	1
Roma	1	1	1	1	1	1	1	1	1		1	1
Napoli	1	1	1	1	1	1	1			1	1	1
Palermo	1	1	1	1	1			1		1		
Torino	1	1	1	1	1	1	1	1	1	1		
Bologna	1	1	1	1		1	1			1	1	1
Genova	1	1	1	1	1	1	1	1	1	1		
Totale	8	8	8	8	7	7	7	6	5	5	5	5

Tab. 4: ordinamento delle relazioni frequenti

Rappresentando in questo modo i dati, non solo abbiamo ordinato i predicati per frequenza, ma soprattutto abbiamo rappresentato l'intero nostro database di transazioni

(tab. 1.1), scartando tutte le relazioni infrequenti e perciò inutili al fine di individuare patterns (o itemsets se utilizziamo l'Apriori) frequenti.

Sfruttando quest'ordinamento, possiamo creare sulla tabella delle transazioni un nuovo FP-Tree.

Per semplificare la rappresentazione, per ogni relazione definita frequente, assegniamo, come già fatto nel caso dei predicati non ripetuti, una lettera dell'alfabeto (in questo modo sarà anche più evidente il trattamento differenziato dei predicati unari da quelli ripetuti).

<Meet, Autostrade>	**8**	**MA**	**A**
<Overlap, SuperStrade>	**8**	**OSS**	**B**
<Contains, Strutture >	**8**	**CnS**	**C**
<Overlap, Ferrovie>	**8**	**OF**	**D**
<Meet, Strutture>	**7**	**MS**	**E**
<Overlap, Fiumi e Laghi>	**5**	**OFL**	**F**
<Meet, SuperStrade>	**5**	**MSS**	**G**

Predicati non ripetuti

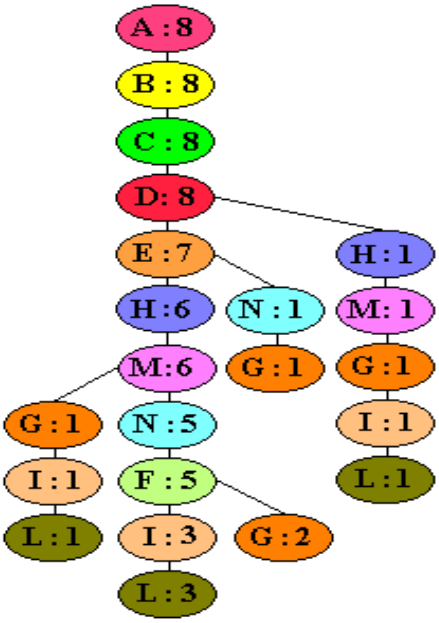
<Meet,Autostrade> <Meet,Autostrade>	**7**	*A2*	**H**
<Overlap,SuperStrade> <Overlap,SuperStrade>	**5**	*B2*	**I**
<Overlap,SuperStrade> <Overlap,SuperStrade> <Overlap,SuperStrade>	**5**	*B3*	**L**
<Overlap,Ferrovie> <Overlap,Ferrovie>	**7**	*D2*	**M**
<Contains,Strutture> <Contains,Strutture>	**6**	*C2*	**N**

Predicati Ripetuti

Basandoci sulla tabella 4, possiamo costruire il FP-Tree relativo:

1	**ABCDEHMNFIL**
2	**ABCDEHMNFIL**
3	**ABCDEHMNFIL**
4	**ABCDEHMGIL**
5	**ABCDENG**
6	**ABCDEHMNFG**
7	**ABCDHMGIL**
8	**ABCDEHMNFG**

Database codificato



FP-Tree con predicati ripetuti

Per prima cosa, possiamo notare che i primi quattro nodi dell'albero (partendo dalla radice), sono uguali per ogni possibile percorso. Ciò ci permette di suddividere l'albero in due sotto-strutture:

- *FP-tree single-path*, costituito dai nodi *A-B-C-D*
- *FP-tree multi-path*, formato dal restante albero.

I due sottoalberi possono essere analizzati separatamente, riducendo lo spazio in cui generare i pattern frequenti.

L'insieme dei k-patterns, generati visitando il *FP-tree single-path*, è:

1-Pattern	Freq	2-Pattern	freq	3-Pattern	Freq
D	**8**	**CD**	**8**	**BCD**	**8**
C	**8**	**BD**	**8**	**ABD**	**8**
B	**8**	**AD**	**8**	**ACD**	**8**
A	**8**	**BC**	**8**	**ABC**	**8**
		AC	**8**		
		AB	**8**		

k-pattern generati dal single-path

L'insieme dei patterns generati dal *multi-path* è:

1-Pattern	freq	2-Pattern	freq	3-Pattern	freq	4-Pattern	freq	5-Pattern	freq
L	5	IL	5	MIL	5	HMIL	5	EHMFN	5
I	**5**	*GL*	*2*	**HML**	**5**	**EHFN**	**5**		
G	**5**	**ML**	**5**	HIL	5	**HMFN**	**5**		
F	**5**	**HL**	**5**	**HMI**	**5**	**EHMN**	**5**		
N	**6**	*EL*	*4*	**MFN**	**5**	**EHFN**	**5**		
M	**7**	*GI*	*2*	**HMN**	**5**				
H	**7**	**MI**	**5**	**HFN**	**5**				
E	**7**	**HI**	**5**	**EHN**	**5**				
		EI	*4*	**EMN**	**5**				
		NI	*3*	**EFN**	**5**				
		FI	*3*	**HMF**	**5**				
		MG	*4*	**EHF**	**5**				
		HG	*4*	**EMF**	**5**				
		EG	**5**	**EHM**	**6**				
		NG	*3*						

	FG	*2*
	NF	**5**
	MF	**5**
	HF	**5**
	EF	**5**
	MN	**5**
	HN	**5**
	EN	**6**
	HM	**7**
	EM	**7**

k-pattern generati dal multi-path

Al momento di generare i k-patterns frequenti, dobbiamo tenere conto di un nuovo vincolo (non presente nel caso delle coppie non ripetute), che ci permette di eliminarne alcuni pur essendo frequenti. A causa della presenza dei predicati ripetuti, infatti, non avrebbe senso individuare un pattern contenente al suo interno predicati relativi alla stessa coppia <*relazione,classe*> e che si differenziano solo per il fatto che sono differenti ripetizioni di essa. I k-patterns aventi al loro interno i predicati *I* e *L* relativi alla coppia <*Overlap,SuperStrade*> (evidenziati in azzurro), vengono perciò scartati in quanto pur essendo frequenti, non ci forniscono informazioni aggiuntive utili.

<*Meet, Autostrade*>	**A**	**H**
<*Overlap, SuperStrade*>	**B**	**I**
<*Overlap, SuperStrade*>	**B**	**L**
<*Overlap, SuperStrade*>	**I**	**L**
<*Contains, Strutture* >	**C**	**N**
<*Overlap, Ferrovie*>	**D**	**M**

Tabella dei predicati incompatibilli

Terminata la generazione dei k-patterns relativi ai due sub FP-Tree e facendo il prodotto cartesiano dei due insiemi ottenuti, determineremo l'insieme di tutti i k-patterns frequenti, i cui items appartengono sia al primo che al secondo albero.
Unendo i tre insiemi di patterns ottenuti, otterremo l'insieme dei k-patterns generabili (con k qualsiasi).

Nel nostro esempio dei due insiemi di k-patterns generati visitando le sottoporzioni del FP-Tree di partenza, dovremmo fare il prodotto cartesiano per ottenere l'insieme dei k-patterns che contengono items appartenenti ai due sottogruppi {A, B, C, D} e {E, H, M, N, F, I, L}. Anche in questo caso, dobbiamo tenere conto del vincolo sui predicati ripetuti, non generando k-pattern aventi al loro interno predicati incompatibili.

4.6 Costruzione delle regole associative spaziali

In quest'ultima fase del procedimento, si genereranno le regole associative spaziali a partire dai k-predicati, o itemsets frequenti se abbiamo utilizzato l'algoritmo Apriori, frequenti.

Prendendo come esempio il 3-pattern frequente "*HMN*" avente supporto 5. Esso esprime che 5 grandi città sulle 8 prese in esame, pari al 62,5%, incontrano (*meet*) 2 autostrade, sono attraversate (*overlap*) da 2 linee ferroviarie e contengono (*contains*) al loro interno 2 strutture atte al commercio e/o all'industria (porti, stazioni, zone industriali, ecc.).

<*Meet,Autostrade*> <*Meet,Autostrade*>	**7**	**H**
<*Overlap,Ferrovie*> <*Overlap,Ferrovie*>	**7**	**M**
<*Contains,Strutture*> <*Contains,Strutture*>	**6**	**N**

Predicati Ripetuti

Ricordando che una regola associativa è un'implicazione tra due congiunzioni di predicati e che ogni predicato possiede l'item implicito "*is_a(x, City)*" che seleziona gli

oggetti della classe referente, possiamo generare tutte le possibili regole associative, semplicemente ricombinando insieme i 4 predicati.

$x \rightarrow H \wedge M \wedge N$	is_a(x, City)→ <Meet,Aut.>2 ∧ <Overlap,Fer.>2 ∧ <Contains,Str.>2
$H \rightarrow x \wedge M \wedge N$	<Meet,Aut.>2→ is_a(x, City) ∧ <Overlap,Fer.>2 ∧ <Contains,Str.>2
$M \rightarrow x \wedge H \wedge N$	<Overlap,Fer.>2→ is_a(x, City) ∧ <Meet,Aut.>2 ∧ <Contains,Str.>2
$N \rightarrow x \wedge H \wedge M$	<Contains,Str.>2→ is_a(x, City) ∧ <Meet,Aut.>2 ∧ <Overlap,Fer.>2
$x \wedge H \rightarrow M \wedge N$	is_a(x, City) ∧ <Meet,Aut.>2→ <Overlap,Fer.>2 ∧ <Contains,Str.>2
$x \wedge M \rightarrow H \wedge N$	is_a(x, City) ∧ <Overlap,Fer.>2→ <Meet,Aut.>2 ∧ <Contains,Str.>2
$x \wedge N \rightarrow H \wedge M$	is_a(x, City) ∧ <Contains,Str.>2→ <Meet,Aut.>2 ∧<Overlap,Fer.>2
$H \wedge M \rightarrow x \wedge N$	<Meet,Aut.>2 ∧ <Overlap,Fer.>2→ is_a(x, City) ∧ <Contains,Str.>2
$H \wedge N \rightarrow x \wedge M$	<Meet,Aut.>2 ∧ <Contains,Str.>2→ is_a(x, City) ∧ <Overlap,Fer.>2
$M \wedge N \rightarrow x \wedge H$	<Overlap,Fer.>2 ∧ <Contains,Str.>2→ is_a(x, City) ∧ <Meet,Aut.>2
$x \wedge H \wedge M \rightarrow N$	is_a(x, City) ∧ <Meet,Aut.>2 ∧ <Overlap,Fer.>2→<Contains,Str.>2
$x \wedge H \wedge N \rightarrow M$	is_a(x, City) ∧ <Meet,Aut.>2 ∧ <Contains,Str.>2→<Overlap,Fer.>2
$x \wedge M \wedge N \rightarrow H$	is_a(x, City) ∧ <Overlap,Fer.>2 ∧ <Contains,Str.>2→ <Meet,Aut.>2
$H \wedge M \wedge N \rightarrow x$	<Meet,Aut.>2 ∧ <Overlap,Fer.>2 ∧ <Contains,Str.>2→ is_a(x, City)

Insieme di regole associative generabili dal pattern HMN

Dopo aver generato tutte le regole si passa alla fase di validazione verificando, per ognuna di esse, se supera il vincolo di minima confidenza imposto. Come precedentemente accennato, il valore del supporto di una regola è calcolabile a partire dai supportri degli items che la compongono, mediante la formula:

$$\text{confidenza}(A \rightarrow B) = \text{Prob}\{B|A\} = \frac{\text{Prob}\{A \cup B\}}{\text{Prob}\{A\}} = \frac{\text{supporto}(A, B)}{\text{supporto}(A)}$$

Poiché il valore di “supporto(A, B)” è il supporto del k-predicato genitore, nel nostro caso “*HMN*”, è sufficiente andare a determinare solamente il supporto relativo al k-

predicato che esprime solo la conseguenza dell'implicazione (supporto(B)); ma poiché B è un predicato più ristretto e generale del nostro, certamente anch'esso è frequente (in base alla proprietà di antimonotonia delle regole associative) e quindi è presente nella tabella dei k-predicati frequenti.
Ipotizzando che la soglia di minima confidenza consentita sia del'80% avremo che 7 regole associative spaziali su 12 sono "forti".

S.A.R.s	Supporto	%
x → H ∧ M ∧ N	5/8	63%
H→ x ∧ M ∧ N	5/7	71%
M → x ∧ H ∧ N	5/7	71%
N → x ∧ H ∧ M	5/6	83%
x ∧ H → M ∧ N	5/7	71%
x ∧ M → H ∧ N	5/7	71%
x ∧ N → H ∧ M	5/6	83%
H ∧ M → x ∧ N	5/7	71%
H ∧ N → x ∧ M	5/5	100%
M ∧ N → x ∧ H	5/5	100%
x ∧ H ∧ M → N	5/7	71%
x ∧ H ∧ N → M	5/5	100%
x ∧ M ∧ N → H	5/5	100%
H ∧ M ∧ N → x	5/5	100%

Valutazione delle regole associative generate

4.7 La valutazione dei costi.

Per valutare i costi del procedimento enunciato, analizziamo distintamente ogni singola fase.

Supponiamo che ***N*** sia il numero delle città che verificano la query, ***L*** il numero di layers del GIS in esame e ***R*** il numero di relazioni possibili.

- *Fase 1: visita degli R+-Tree.*

Abbiamo ipotizzato l'uso di un R+-Tree per indicizzare lo spazio di un layer. Dovremo, quindi, fare $\boldsymbol{N}$ visite per ogni R+-Tree, cioè $\boldsymbol{N*L}$ visite totali. Per ogni oggetto referente avremo una riga (transazione) nella tabella 1 e per ogni layers una colonna; la dimensione di tale tabella sarà $\boldsymbol{N*L*k_{i,j}}$ *dove* $\boldsymbol{k_{i,j}}$ indica il numero di coppie <*relazione,oggetto*> che la città *i* ha con il layer *j*.

- *Fase 2: Creazione della tabella delle relazioni inter-layers.*

In questa fase astraiamo le coppie presenti nella tabella 1, sostituendo gli oggetti con la classe (o l'eventuale sottoclasse se desideriamo una specializzazione) a cui appartengono.
Questa fase consiste, perciò, in una semplice visita della tabella delle transazioni (tabella 1), in una identificazione dell'oggetto (per capire a quale gruppo appartiene) e nell'aggiornamento di tale tabella o, se vogliamo mantenere queste informazioni, nell'aggiornamento di una nuova tabella di dimensione paragonabile a quella della tabella 1. Il tutto avviene in $\boldsymbol{O(N)}$ tempo. Qualora i dati ricavati nella visita degli R+-Tree non fossero utili, si potrebbe accorpare le fasi 1 e 2 inserendo direttamente la coppia <*relazione,classe*>, generata nel passo di astrazione.

- *Fase 3: calcolo delle frequenze.*

In questo passo si visita la tabella 1.1 delle coppie <*relazione,classe*> (tempo $\boldsymbol{O(N)}$) e si aggiorna la tabella delle frequenze. Tale tabella avrà una riga per ogni transazione esistente e un numero di righe pari al numero di coppie possibili ($\boldsymbol{R}$ diverse relazioni per ogni layer). La dimensione sarà perciò $\boldsymbol{N*L*R}$. Poiché ogni cella esprimerà la presenza o meno di una data coppia in una precisa transazione, è sufficiente che la tabella abbia un solo bit per ogni cella (1 se la relazione esiste, 0 altrimenti).

- *Fase 4: raccolta degli items frequenti.*

Scandendo ogni singola colonna e sommando i valori in essa contenuti si otterrà la frequenza del predicato relativo; confrontando tale valore con il minimo supporto si scoprirà se tale coppia è frequente o meno. In questa fase è perciò richiesta una scansione completa della tabella 2 (in tempo $\boldsymbol{N*L*R}$), un confronto e un eventuale inserimento del predicato in un vettore degli items frequenti.

- *Fase 5: generazione degli itemsets frequenti.*

Il costo di questa fase è strettamente dovuto al tipo di algoritmo che si sceglie di utilizzare per determinare quali itemsets sono frequenti (Apriori o FP-Tree). L'uso dell'FP-Tree implica una scansione della tabella 1.1 per ordinare gli items frequenti, presenti in ogni transazione e cancellare quelli inutili e la costruzione dell'albero. Per elaborazioni aventi un numero elevato di transazioni è certamente consigliato l'uso di tale metodo, mentre per generazioni parziali di itemsets (cioè solo fino ad una lunghezza prefissata), si consiglia l'uso dell'algoritmo Apriori.

Qualora si intendesse ricercare relazioni anche con predicati ripetuti, è necessario sviluppare il processo precedente, ampliando alcune fasi e introducendo alcune strutture dati di supporto.

Le fasi di ricerca e memorizzazione delle coppie (fasi 1 e 2) rimangono invariate, mentre nella fase 3 (calcolo delle frequenze) la tabella generata (tabella 2) dovrà contenere anche il numero di volte in cui la coppia *j* appare nella transazione *i*. In questo caso la tabella non avrà solo un bit per cella, ma ognuna di esse dovrà contenere un intero.

Nella fase 4 (raccolta degli items frequenti), dobbiamo verificare per ogni items se è frequente e se esistono sue ripetizioni anch'esse frequenti. Per questo motivo, dobbiamo, al momento della visita della tabella 2, aggiungere una nuova tabella per memorizzare la frequenza di tutte le ripetizioni che avrà dimensione $\boldsymbol{N * Max}$, dove ***Max*** è il numero massimo di volte che una coppia appare in una transazione. Si creerà, inoltre, una nuova tabella di dimensione $\boldsymbol{N * RF}$ (dove ***RF*** indica il numero di relazioni frequenti, comprese le ripetizioni), avente un bit per ogni cella per indicare se il predicato (espresso dalla colonna) appare o no nella transazione (espressa dalla riga). In questo modo, otterremo un database di transazioni aventi tutte items frequenti e ordinati per supporto decrescente, che è la condizione essenziale su cui creare un FP-Tree.

Al momento della generazione dei k-predicati, poi, bisogna mantenere in memoria una piccola tabella che esprime l'incompatibilità di due predicati (dovuta al fatto che sono ripetizioni differenti della stessa coppia <*relazione,classe*>).

Capitolo 5
Implementazione del metodo

In questo capitolo verranno presentate le scelte progettuali che hanno guidato tutto il processo di sviluppo dell'applicazione realizzata.

Lo scopo finale della nostra implementazione è quello di progettare un algoritmo per la determinazione di regole associative spaziali tra gli oggetti presenti nei diversi layers inseriti in un sistema geografico.

Nella prima parte si decriverà l'ambiente di lavoro utilizzato e la libreria sulla quale è basata tale l'implementazione; nella seconda parte, invece, mostrerò in dettaglio i componenti del metodo proposto.

5.1 Ambiente di lavoro

Comincerò col descrivere la libreria dalla quale ho preso spunto per la mia sperimentazione.

L'architettura del "*Java Unified MappingPlatform*" (JUMP) è una GUI per la programmazione di processi sui dati spaziali, realizzata interamente in Java dal gruppo di lavoro "*JUMP Project"*. **[JUMP]**

L'architettura del Jump System è organizzata in due livelli:

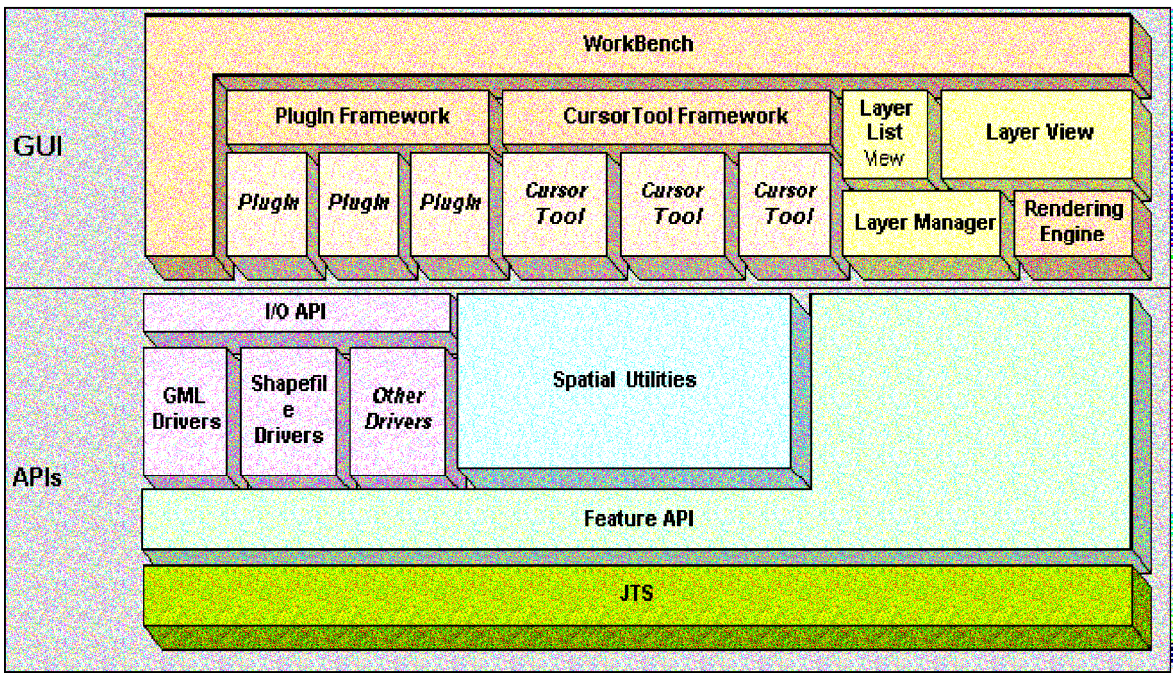
1. Jump APIs: è un insieme modulare di APIs Java in grado di supportare i dati spaziali di I/O, le features, le collezioni e gli algoritmi fondamentali come, ad esempio, quelli di "Spatial Access Methods". Queste API sono implementate su Java Topology Suite (JTS) che fornisce i modelli geometrici, tra cui il 9-intersection model di Egenhofer utilizzato come standard nella scelta dei predicati spaziali che andranno a comporre le regole associative, e le operazioni geometriche basilari. Lo

JTS non è considerato essere parte di JUMP, ma è comunque utilizzato da i suoi più comuni componenti.

2. Jump Workbench: è un'applicazione interattiva che fornisce un'iterfaccia utente alle funzionalità delle APIs di JUMP. Il suo scopo principale è quello di mediare tra l'utente e le funzionalità spaziali offerte e di visualizzarne i risultati.

Ognuno di questi due packages è strutturato in modo tale da essere il più possibile estensibile, modulare e riusabile. Essi fanno affidamento, per questo, sull'affidabilità e il dinamismo della piattaforma *java* che li supporta.

La figura seguente mostra i maggiori componeti dell'architettura JUMP.



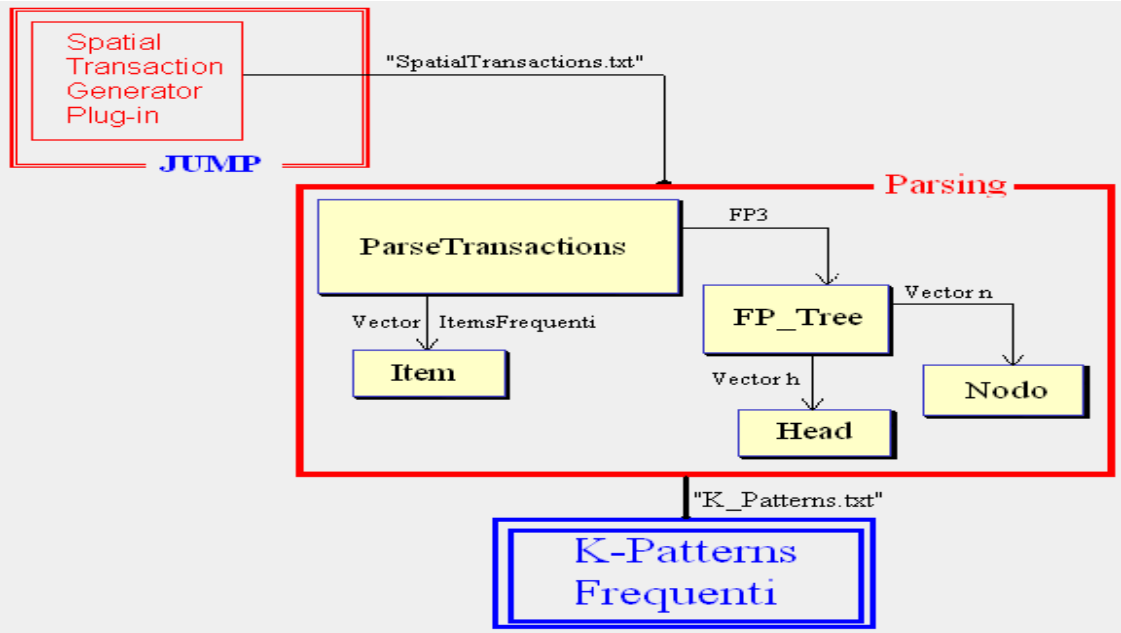
Architettura JUMP

5.2 Applicazione del metodo

L'implementazione del metodo è realizzata creando un plug-in del GIS JUMP che, analizzando i layer forniti, restituisce tutte le coppie *<relazione,oggetto_referenziato>* relative ad ogni singolo oggetto referente. Tramite una successiva fase di analisi, che comporterà anche la creazione e la visita di un FP-Tree, da tali coppie sarà possibile generare gli itemsets frequenti.

Il metodo può essere diviso in due componenti distinte:

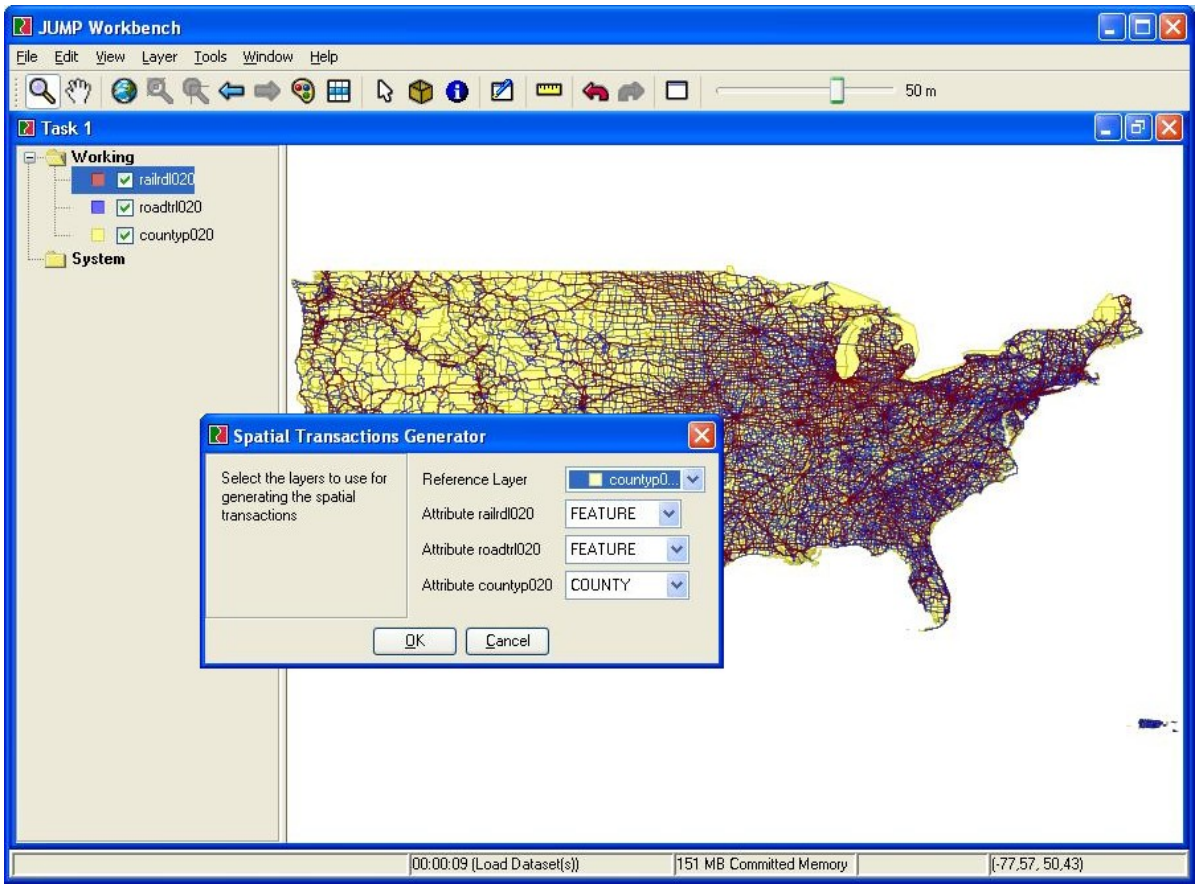
- un plug-in del GIS JUMP, che funge da interfaccia tra l'algoritmo di knowledge discovery e il sistema JUMP, necessario per la raccolta delle relazioni spaziali intercorrenti tra gli oggetti della classe di riferimento e quelli presenti nelle classi referenziate, e
- l'algoritmo di parsing vero e proprio, che produrrà i k-predicati frequenti necessari alla costruzione delle regole associative spaziali mediante l'uso di un Frequent Pattern Tree.



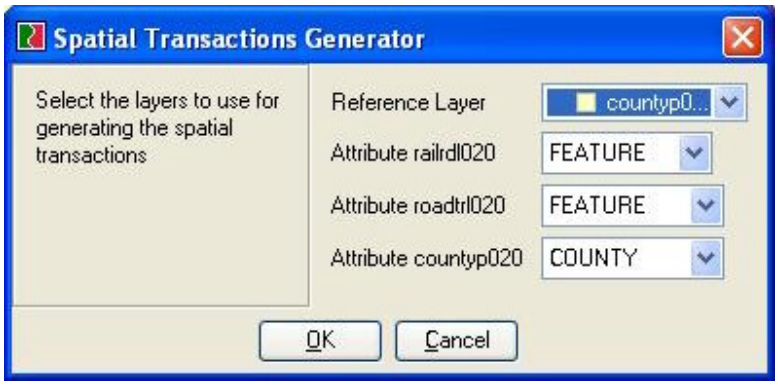
Relazioni tra le classi dell'applicazione

52.1 Il plug-in di generazione del database delle transazioni

Questo componente, sviluppato come plug-in del GIS JUMP, è costituito da una interfaccia visuale che, tramite una finestra, permette all'utente di scegliere tra i layers selezionati per l'analisi quello di riferimento e per i restanti, che formeranno l'insieme dei layers referenziati, permette di specificare l'attributo principale.



Screenshot dell'interfaccia di JUMP



Screenshoot del plug-in

Avviato il processo di generazione delle transazioni (richiamando il metodo *generateSpatialTransaction()* del plug-in), si ricercano per ogni layer referenziato le relazioni esistenti tra gli oggetti che appartengono al layer in esame e che si trovano in prossimità dell'oggetto referente. Il concetto di "*prossimità*" è espresso tramite il metodo dell'oggetto referente *buffer(x)* definito in JUMP; esso restituisce l'insieme degli oggetti presenti in un intorno circolare di raggio *x*. Non appena un oggetto va a cadere all'interno, o quantomeno tocca tale intorno, si passa alla fase di determinazione della relazione spaziale intercorrente tra i due oggetti in esame (ovvero tra l'oggetto referente e l'oggetto appena trovato). In questa fase si analizza la matrice delle intersezioni definita nello standard del nine-intersection matrix di Egenhofer.

Una volta determinata la relazione si memorizza in un semplice file di testo, *SpatialTransaction.txt*, la coppie *<relazione_spaziale,oggetto_referente>* trovata.

```
protected void generateSpatialTransaction(Layer training,
                                          Hashtable attributeMapping,
                                          Vector layers,
                                          PrintWriter out) {
System.out.println("LAYER"+training.getName());
String refAttr = (String) attributeMapping.get(training.getName());
for (Iterator i = layers.iterator(); i.hasNext();) {
        Layer l = (Layer) i.next();
        String attr = (String) attributeMapping.get(l.getName());

//      salvataggio nome layer
        out.println(" **** Layer " + l.getName() + " **** ");
        System.out.println(" **** Layer " + l.getName() + " **** ");
        int tot = 1;
        IndexedFeatureCollection ifc = new IndexedFeatureCollection(
```

```
                                                    l.getFeatureCollectionWrapper());
        for (Iterator j=training.getFeatureCollectionWrapper().iterator(); j.hasNext();) {
                Feature f = (Feature) j.next();
                Geometry rg = f.getGeometry().buffer(0.03d);

//              salvataggio nome oggetto referente
                out.println(f.getAttribute(refAttr).toString());
                System.out.println(tot++);

                List lst = ifc.query(rg.getEnvelopeInternal());
                for (Iterator k = lst.iterator(); k.hasNext();) {
                        Feature f1 = (Feature) k.next();
                        Geometry g1 = f1.getGeometry();

                        if (g1.getEnvelopeInternal().intersects(rg.getEnvelopeInternal())) {
                                IntersectionMatrix im = rg.relate(g1);
                                String relation = null;
                                if (im.isContains())
                                        relation = "contains";
                                else if (im.isCrosses(rg.getDimension(), g1.getDimension()))
                                        relation = "crosses";
//                              else if (im.isDisjoint())
//                                      relation = "disjoint";
                                else if (im.isIntersects())
                                        relation = "intersects";
                                else if (im.isOverlaps(rg.getDimension(), g1.getDimension()))
                                        relation = "overlaps";
                                else if (im.isTouches(rg.getDimension(), g1.getDimension()))
                                        relation = "touches";
                                else if (im.isWithin())
                                        relation = "within";

                                if (relation != null) {

//                                  salvataggio coppia <relazione, classe & oggetto_referenziato>
                                    out.println("    (" + relation + ", "
                                                    + l.getName() +":\""
                                                    + f1.getAttribute(attr).toString().trim()+ "\")");
                                                    }
                                }
                        }
                }
        }
}
```

plugin.generateSpatialTransaction()

5.2.2 Il processo di parsing

Nella seconda parte dell'applicazione, si analizza il file di testo, generato dal plug-in appena descritto, al fine di determinare quali items siano da considerare frequenti. In seguito, si genereranno gli itemsets frequenti, o k-predicati, che fungeranno da generatori nella fase di definizione delle regole associative spaziali.

In base a quanto enunciato nel capitolo 4, il parsing dei dati spaziali raccolti, può essere spezzato in 5 differenti fasi:

1) Creazione della tabella delle transazioni (*createTable1_1()*)
2) Creazione della tabella degli items con ripetizioni (tabella 3) e determinazione degli items frequenti (*createTable_3()*).
3) Ordinamento degli items determinati frequenti al passo precedente e rielaborazione della tabella delle transazioni al fine di eliminare sia gli items infrequenti che tutte le ripetizioni infrequenti di items frequenti (*OrdinamentoTabella1_2()*).
4) Creazione del FP_Tree con ripetizioni (*CreaFP_Tree()*).
5) Visita del FP_tree per determinare i k-predicati frequenti (*FP_Growth()*).

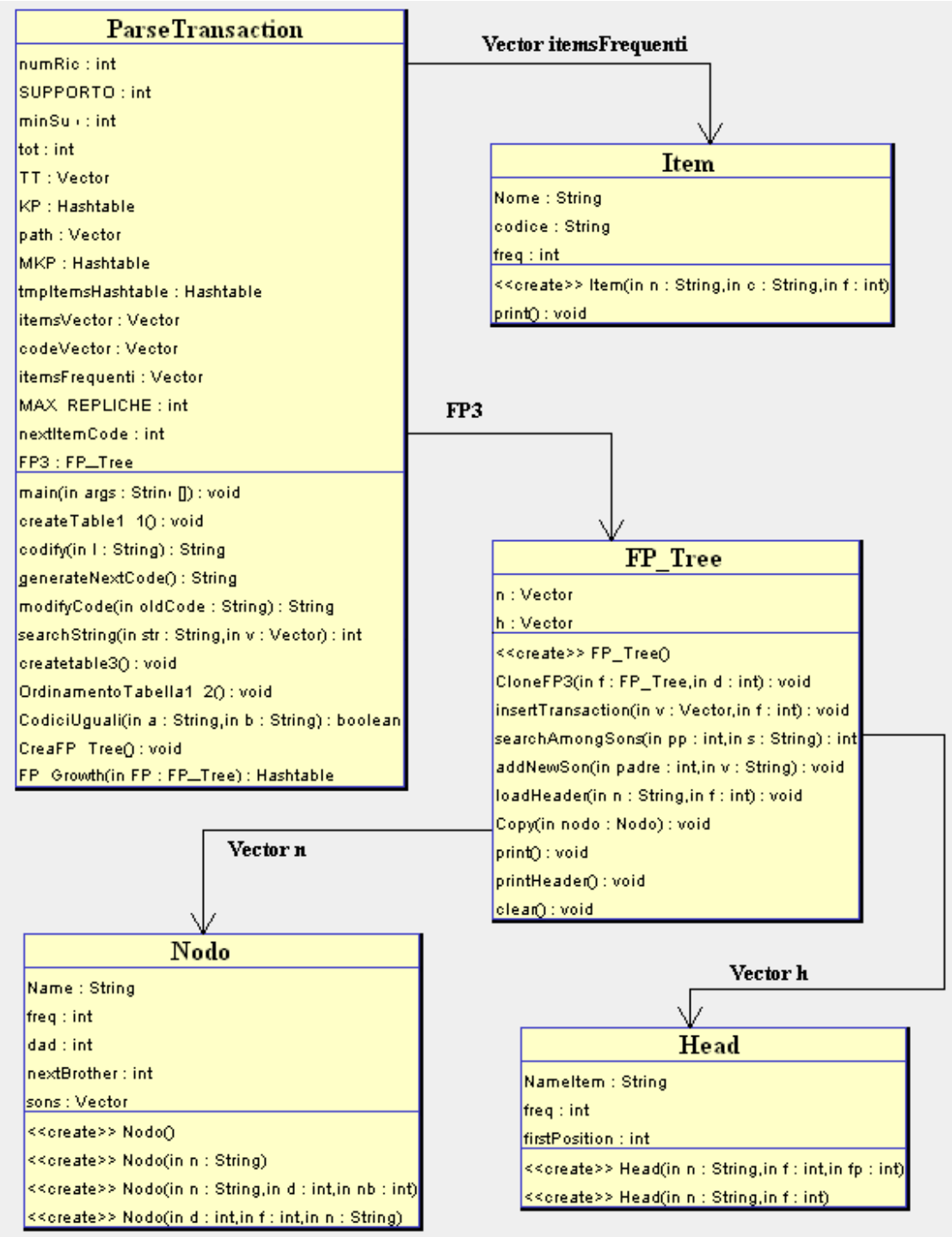


Diagramma UML del package “parsing”

5.2.2.1 Creazione della tabella delle transazioni: metodo createTable1_1

Il file di testo generato dal plug-in, riporta layer per layer tutte le terne <*oggetto referente, relazione spaziale, oggetto referenziato*> riscontrate.

Compito di questa fase è quella di determinare per ogni oggetto referente, tutte le coppie < *relazione spaziale, oggetto referenziato* > ad esso relative.

Il parsing del file di testo sfrutta la struttura sintattica con cui viene creato; infatti, in tale file di testo si possono riscontrare soltanto 3 tipi diversi di righe:

1- Quelle che iniziano con "****" che indicano l'inizio dei dati relativi ad un layer.
2- Quelle che iniziano con un carattere alfabetico, che rappresenta il valore dell'attributo di riferimento che un oggetto referente possiede (ad esempio il *nome* o il suo *codice identificativo*).
3- E quelle che iniziano con " (" che indicano una coppia <relazione, oggetto > relativa all'ultimo oggetto referente inserito.

Esempio della sintassi del file:

```
**** Nome Layer ****
oggetto referente
   ( relazione spaziale, oggetto referenziato)
   ( relazione spaziale, oggetto referenziato)
   ( relazione spaziale, oggetto referenziato)
oggetto referente
   ( relazione spaziale, oggetto referenziato)
……
```

In questa fase, si analizza il file di testo, generato dal plug-in di Jump, riga per riga e si genererà la tabella delle transazioni:

- se una riga comincia con una sequenza di quatto asterischi "****", si indica l'inizio di un layer, ovvero che tutte le righe successive (finché non si incontrerà una nuova riga con i quattro asterischi, oppure finchè non si giungerà alla fine del file), rappresentano le relazioni presenti in quel dato layer.
- se la riga inizia con " (", allora rappresenta una coppia <*relazione,oggetto_referenziato*>; in questo caso, la coppia deve essere memorizzata nella riga della tabella delle transazioni relativa all'oggetto referente corrente. Per ragioni di compattezza, invece di memorizzarla direttamente in tale transazione, la sottoporremo a due fasi di codifica:

  la prima (metodo "*codify()*"), verificherà se tale coppia non è mai stata incontrata prima d'ora, in tal caso gli si assegnerà un nuovo codice identificativo alfanumerico che la rappresenta in maniera più compatta. Viceversa, se la coppia è già nota, allora si utilizzerà il codice precedentemente generato per essa. In questo modo ad ogni possibile coppia verrà assegnato un codice.

  La seconda fase di codifica, espressa dal metodo "*searchString()*", verifica se il codice ricavato dalla codifica precedente è già presente nella transazione corrente; in questo caso, siamo in presenza di un doppione dello stesso item. Sfruttando questa caratteristica, possiamo, aggiornando opportunamente il codice dell'item già presente nella transazione corrente, non inserire questo nuovo oggetto risparmiando spazio. Infatti, il codice identificativo di un dato item si compone di tre parti: un codice numerico progressivo, il carattere separatore "#" e un altro codice numerico; il primo codice è l'identificativo dell'item ottenuto col passo di codifica precedente, mentre il secondo codice indica il numero di volte che tale item è presente nella transazione corrente.

  es. <*contains, railroad*> → *11#1*

  <*intersect,lake*>,<*intersect,lake*>,<*intersect,lake*> → *13#3*

  In questo modo, se il codice dell'item in esame è già presente nella transazione corrente, è sufficiente incrementare di 1 il secondo codice alfanumerico per ottenerne l'inserimento.

- Se una riga non rientra in nessuno dei suddetti casi, indicherà, salvo casi degeneri come la riga vuota, necessariamente un nuovo oggetto referente. In questo caso, per come è

strutturato il file di testo, è possibile che tale oggetto sia già stato incontrato al momento di analizzare i dati relativi ad un layer precedente. Se così fosse, tutte le coppie sottostanti a questa riga e relative perciò a questo oggetto referente, non devono essere inserite in una nuova transazione ma devono essere accodate alla transazione, già presente nella tabella, relativa a tale oggetto. Questo problema, viene risolto utilizzando una tabella hash in cui memorizzo tutti gli oggetti referenti che incontro. Se l'oggetto non appare in tale *hashtable* sarà certamente nuovo e si creerà perciò una nuova riga nella tabella.

Le strutture dati utilizzate in questa fase sono:

- Una matrice dinamica (un vector() di vector()), detta *TT* che rappresenta la *Transaction Table* o tabella 1.1 del processo di knowledge discovery esposto al capitolo 4.
- Un vettore *TMPCode* che memorizza il codice dell'item corrente.
- Un puntatore intero *pos* che indicherà la transazione corrente di *TT*.
- Una Hashtable *NOR* (Number_of_record) che per ogni oggetto referente, mi indica la posizione nel vettore *TT* della transazione relativa.
- Una Hashtable *tmpItemsHashtable* che per ogni item ne memorizza il relativo codice. Poiché questa struttura viene utilizzata esclusivamente per verificare se un item è già stato incontrato (nel qual caso si restituirà il relativo codice), questa hashtable viene azzerata ogni qual volta si analizza un nuovo layer, in quanto tale item certamente non può essere stato individuato in un layer differente, dato che, come precedentemente detto, un item è costituito dalla coppia <*relazione,oggetto_referenziato*> e un oggetto si trova esclusivamente nel suo layer di riferimento. Sfruttando tale principio, si mantiene bassa la dimensione di tale struttura, permettendo anche ricerche più veloci.
- Due stringhe "*line*" e "*line2*" necessarie per il parsing del file di testo la prima, e per eliminare operazioni di ricerca ed inserimento relative ad oggetti non validi, quali ad esempio oggetti referenti non aventi items, la seconda.

5.2.2.2 Ricerca degli items frequenti: metodo createtable3

In questa fase, si analizza la tabella delle transazioni *TT* e si crea la tabella 3 definita nel paragrafo 4.5.

Per ogni transazione presente in *TT*, si prendono tutti i codici degli items in essa contenuti e si aggiorna la tabella 3. Come precedentemente detto, tale tabella conterrà per ogni item e per ogni sua ripetizione il numero di volte che esso appare nella tabella delle transazioni.

Ad es. se una transazione ha i seguenti codici: "*1#2, 3#1, 4#3, 2#1*" allora si aggiornerà la tabella 3 incrementando di 1:

- nella riga 1, le colonne 1 e 2,
- nella riga 3, la colonna 1,
- nella riga 4, le colonne 1, 2 e 3
- e nella riga 2, la colonna 1.

In questo modo, con una semplice visita della tabella, possiamo evidenziare quali items, ed eventualmente quali loro ripetizioni, sono frequenti confrontando i valori presenti nelle varie colonne con la soglia di minimo supporto richiesto.

Caratteristiche peculiari di questa tabella sono che:

- La prima colonna rappresenta il numero di volte in cui gli items appaiono nella tabella delle transazioni tralasciando le ripetizioni. In questo modo, se si volessero ricercare itemsets senza ripetizioni è sufficiente analizzare solo la prima colonna per determinare gli items frequenti.
- I valori presenti in una colonna possono essere al più uguali a quelli presenti nella colonna precedente; questo perché il numero di k ripetizioni di un item non può essere maggiore del numero di k-1 ripetizioni dello stesso item. Per questa ragione, se la n-esima ripetizione di un item è infrequente (il valore presente nella colonna n relativa ad un dato item è inferiore al minimo supporto), allora n+1 ripetizioni dello stesso sono infrequenti; quindi, per ogni item (e quindi per ogni riga), non è

necessario verificare se tutte le sue ripetizioni superano la soglia: basta trovarne una che non la supera e tutte le colonne successive certamente non saranno frequenti.

Terminata la fase di inserimento, si passa ad analizzare la tabella 3 appena creata utilizzando la proprietà sopra enunciata, valutando quali items hanno un supporto alto e inserendoli nel vettore degli items frequenti *itemsFrequenti*. Da notare che gli items inseriti in questo vettore anche se sono relativi alla stessa coppia <*relazione,oggetto_referenziato*> e differiscono solo per il numero di ripetizioni (il codice dopo il separatore "#"), d'ora in poi verranno trattati come indipendenti, come fossero relativi ad items differenti, salvo poi eliminare in fase di generazione dei k-predicati quelli aventi due o più items "fratelli".

5.2.2.3 Fase di ordinamento: metodo OrdinamentoTabella1_2

In questo modulo si procede ad ordinare gli items frequenti determinati al passo precedente e a selezionare gli items presenti nella tabella delle transazioni, scartando quelli inutili.

Questa fase è necessaria per "preparare" i dati per l'inserimento nel Frequent Pattern Tree (preprocessing delle transazioni).

Innanzi tutto, si ordinano gli items frequenti per frequenza decrescente. In seguito, si analizza la tabella delle transazioni, generata al passo 1, e per ogni transazione si scartano gli items infrequenti e si ordinano i frequenti in base a come appaiono nel vettore degli item frequenti appena ordinato.

La seconda parte è realizzata per verificare, per ogni item presente in una transazione se appare nel vettore degli items frequenti (quindi se è frequente). Se si trova un riscontro, si pone il valore *true* in un vettore di booleani "*BoolTempVect*" nella stessa posizione in cui appare l'item frequente nel vettore *itemsFrequenti*. Terminato il parsing della transazione, si sostituiscono i suoi items con quelli presenti nel vettore *itemsFrequenti* aventi lo stesso indice in cui nel vettore di booleani si riscontra un *true*.

Transazione k-esima di *Transaction Table*										
1#2	3#2	4#4	5#1	6#1	7#2	8#1	9#3	…	…	…

Vettore *itemsFrequenti*

3#1	2#1	3#2	1#1	2#2	4#1	5#1	1#2	3#3	6#1	4#2	1#3	5#2	5#3

Vettore *BoolTempVect*

T	*F*	**T**	**T**	*F*	**T**	**T**	**T**	*F*	**T**	**T**	*F*	*F*	*F*

Transazione k-esima di *Transaction Table* ordinata							
3#1	3#2	1#1	4#1	5#1	1#2	6#1	4#2

Esempio di ordinamento della tabella delle transazioni

5.2.2.4 Generazione del Frequent pattern Tree: metodo CreaFP_Tree

In questa fase si genera il Frequent Pattern Tree corrispondente alla tabella delle transazioni ordinata precedentemente.

Innanzi tutto vediamo le strutture dati che compongono la classe *FP_Tree*.

- un vettore *h* di oggetti di tipo *head* che esprime lo header; quest'ultimo è necessario per memorizzare, per ogni item frequente, la posizione del primo nodo della catena dei nodi presenti nell'albero relativi a tale item. Un oggetto *head* possiede un nome *NameItem*, una frequenza *freq* e un puntatore intero al primo nodo della catena.
- un vettore *n* di oggetti di tipo *nodo* che rappresenta l'albero. Un nodo dell'albero possiede:
  - un nome *Name*
  - una frequenza relativa *freq*

- due puntatori interi: *dad* indica l'indice del nodo padre nel vettore *n* mentre *nextBrother* memorizza il prossimo nodo fratello, ovvero il prossimo nodo nell'albero relativo allo stesso item. Se un indice ha valore "–1" indica che non esiste il nodo successivo (non ha padre o è l'ultimo nodo della catena dei fratelli).
- un vettore di puntatori interi *sons* necessario per collegare un nodo a tutti i suoi figli.

La creazione di un Frequent Pattern tree avviene, sfruttando la classe *FP_Tree*, in due momenti:

- la generazione del header, inserendo nel vettore *h* un oggetto *head* per ogni item frequente e
- la creazione dell'albero vero e proprio "parsando" tutta la tabella delle transazioni ordinata e, per ogni sua transazione, chiamando il metodo *insertTransaction()* del FP_Tree. Questo metodo, partendo dal nodo *root* dell'albero, verifica se esiste un cammino che rappresenti il vettore di items ricevuto come parametro o quanto meno una sua parte; in quest'ultimo caso si inseriranno tanti nuovi nodi nell'albero fino a quando non si completerà il cammino che esprime la transazione da inserire. Inoltre, ogni volta che si visita un nodo dell'albero si incrementa la sua frequenza mentre ogni volta che se ne crea uno nuovo lo si lega alla opportuna catena di fratelli.

5.2.5 Parsing del FP-Tree: metodo FP_Growth

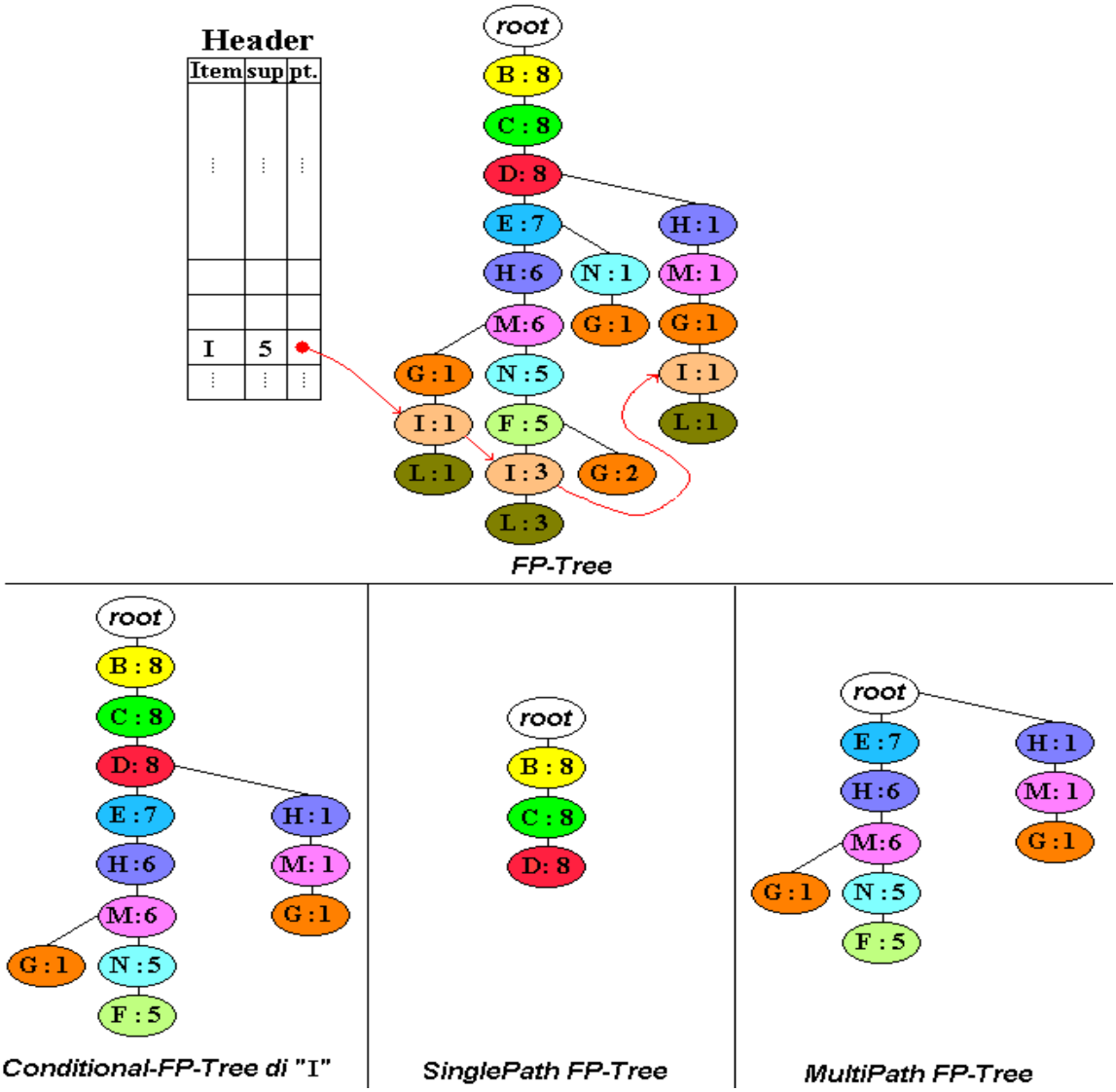
In questa fase, si generano i k-pattern frequenti rappresentati nel FP-Tree definito nel passo precedente.

Il metodo procede ricorsivamente generando, per ogni item presente nel header, il suo personale FP-Tree (*Conditional FP-Tree*), ovvero l'albero avente tutti e soli i nodi che appartengono ai cammini che dalla radice (nodo "*root"*) giungono ai nodi relativi all'item in esame.

Al fine di ridurre la complessità e i costi necessari a elaborare l'intero albero, si cerca, se possibile, di spezzarlo in due sottoalberi distinti:

- un Single-Path FP-Tree e
- un Multi-Path FP-Tree.

Un FP-Tree possiede un *Single-Path FP-Tree* se, a partire dalla radice, esiste un solo cammino possibile, ovvero se esiste una unica sequenza di nodi percorribile. Se così è, lo si estrapola dal "Conditional" e si generano tutti i k-pattern in esso contenuti. I restanti nodi formeranno il *Multi-Path FP-Tree*.



Esempio di suddivisione di un FP-Growth

Su questo nuovo albero si chiama ricorsivamente il metodo in esame, fino a quando non si raggiungerà un livello in cui tale albero sarà vuoto. In questo caso, al ritorno dalla ricorsione si otterrà l'insieme dei k-pattern presenti nel *multi-path* che si andrà ad aggiungere a quello generato dalla visita del *Single-Path*.

Questi due insiemi però, non sono sufficienti a racchiudere la totalità dei k-pattern generabili; infatti, sfruttando il FP-Tree in figura, possiamo ad esempio notare che, il 3-pattern "*B*-*C*-*E*" ha gli items *B* e *C* appartenenti al Single path mentre *E* appartiene al Multi-Path. Questo tipo di itemset lo si può rilevare soltanto facendo il prodotto cartesiano dei due insiemi di k-pattern frequenti generati, ottenendo quello formato da k-patterns aventi items apprtenenti ai due insiemi distinti.

L'unione dei tre insiemi costituisce la totalità dei k-pattern frequenti possibili.

In conclusione perciò, avremo non due, ma ben tre insiemi di k-patterns frequenti per ogni item di partenza.

Poiché però, gli items di cui ci occupiamo non sono tutti indipendenti tra loro, ma anzi, taluni sono relativi alla stessa coppia <relazione,oggetto>, cioè sono ripetizioni diverse dello stesso predicato, tra tutti i k-patterns generati finora, ve ne sono di ridondanti, ovvero sono sequenze di codici, alcuni dei quali si riferiscono allo stesso item e che differiscono solo per il secondo codice (quello che esprime il numero di ripetizioni). Questi k-pattern, non ci forniscono informazioni aggiuntive, quindi devono essere eliminati.

Per capire l'inutilità di tali k-pattern, basti pensare ad una situazione ipotetica in cui si hanno tre items frequenti:

A. <overlap,ferrovia>,
B. <meet,autostrada> e
C. <meet,autostrada>#2.

Essi possono dare origine ai seguenti k-predicati:

A-B, A-C, B-C, A-B-C.

Ipotizzando che tutti e quattro siano frequenti, possiamo notare che:

- i primi due, *A*-*B*, *A*-*C*, ci forniscono informazioni utili,

- il terzo, *B-C*, non ci dice nulla di significativo, in quanto certamente se è frequente che una città "incontra" 2 autostrade certamente sarà frequente che ne incontri solo una.
- L'ultimo predicato poi, pur mettondo in evidenza una relazione tra "ferrovie" e "autostrade", non aggiunge nulla di più di quello che i due predicati *A-B* e *A-C* ci hanno già detto.

I k-predicati che restano al termine di questa fase di epurazione, sono tutti e soli gli itemset validi generabili dal FP-Tree dato e conseguentemente dalla tabella delle transazioni fornita.

Per ottimizzare il tempo di elaborazione, inoltre, si è deciso di anticipare questa fase di selezione dei k-predicati validi, spostandola dal termine della generazione di tutti gli itemsets frequenti, all'interno della fase ricorsiva; in questo modo, non appena si genererà un itemset non valido verrà scartato. Questo ci permette di non generare, al ritorno dalla ricorsione, tutti i successivi predicati non validi (ad esempio se non genero il predicato A-B-C, non posso generare il predicato A-B-C-D).

```
Procedure FP-growth(Tree, α){
(1) if Tree contains a single prefix path
        // Mining single prefix-path FP-tree
(2)      then {
(3)             let P be the single prefix-path part of Tree;
(4)             let Q be the multipath part with the top branching node replaced by a null root;
(5)             for each combination (denoted as β) of the nodes in the path P do
(6)             generate pattern β U α with support = minimum support of nodes in β;
(7)             let freq pattern set(P) be the set of patterns so generated; }
(8)     else let Q be Tree;
(9)     for each item ai in Q do {
                // Mining multipath FP-tree
(10)            generate pattern β = ai U α with support = ai .support;
(11)            construct β's conditional pattern-base and then β's conditional FP-tree Treeβ ;
(12)            if Treeβ != Ø
(13)                    then call FP-growth(Treeβ, β);
(14)                    let freq pattern set(Q) be the set of patterns so generated; }
(15) return     (freq_pattern_set(P) U
                freq_pattern_set(Q)  U
                (freq_pattern_set(P) × freq_pattern_set(Q)))
}
```

Psuedocodice del FP-Growth classico

```
Public static Hashtable FP_Growth(FP_Tree FP){
      numRic++;
      int freq = 0;
      Hashtable allKPath = new Hashtable();
      for (int i=FP.h.size()-1; i>=0; i--){
            Vector tmpTT = new Vector();
            Vector tmpF = new Vector();
            Hashtable tmpIF = new Hashtable();
            Hashtable KSPath = new Hashtable();
            Hashtable KMPath = new Hashtable();
            String ITEM = ((Head) FP.h.get(i)).NameItem;
            int freqItem = ((Head) FP.h.get(i)).freq;
            allKPath.put(ITEM, ""+freqItem);
            int cnt = ((Head) FP.h.get(i)).firstPosition;
            if (numRic == 1){
                  for (Enumeration e = allKPath.keys();
                              e.hasMoreElements(); ){
                        String key = (String) e.nextElement();
                        KP.put(key, (String) allKPath.get(key));
                  }
                  allKPath.clear();
            }
            while (cnt > -1){
                  Vector tmpTrans = new Vector();
                        int pos = ((Nodo) FP.n.get(cnt)).dad;
                  freq = ((Nodo) FP.n.get(cnt)).freq;
                  while (pos >0){
                        int supp = freq;
                        Nodo NodoCorrente = (Nodo) FP.n.get(pos);
                        String Name = (String) NodoCorrente.Name;
                        if (tmpIF.containsKey(Name)){
                              supp += (Integer.valueOf((String)
                                    (tmpIF.get(Name)))).intValue();
                              tmpIF.remove(Name);
                        }
                        tmpIF.put(Name, ""+supp);
                        tmpTrans.addElement(Name);
                        pos = NodoCorrente.dad;
                  }
                  tmpF.addElement(""+freq);
                  tmpTT.addElement(tmpTrans);
                  cnt = ((Nodo) FP.n.get(cnt)).nextBrother;
            }
      FP_Tree newFP3 = new FP_Tree();
            for (Enumeration e = tmpIF.keys(); e.hasMoreElements(); ){
                  Object key = e.nextElement();
                  int supp = (Integer.valueOf((String)
                              (tmpIF.get(key)))).intValue();
                  if (supp < minSup)
                        tmpIF.remove(key);
                  else newFP3.loadHeader((String) key, supp);
            }
```

```
            for (int k=0; k< tmpTT.size(); k++){
                  Vector t = (Vector) tmpTT.get(k);
                  Vector tt = new Vector();
                  int dim = t.size();
                  int supp = (Integer.valueOf((String)
                              tmpF.get(k))).intValue();
                  for (int j=dim-1; j>=0; j--){
                        String key = (String) t.get(j);
                        if (tmpIF.containsKey(key)){
                              tt.addElement(key);
                        }
                  }
                  newFP3.insertTransaction(tt, supp);
            }

            // SINGLE PATH FP_TREE
            Vector SP_FP3 = new Vector();
            Vector SP_freq = new Vector();
            int dad = 0;
            while (((Nodo)newFP3.n.get(dad)).sons.size() == 1 ) {
                  int ind = (Integer.valueOf((String)
                      ((Nodo)newFP3.n.get(dad)).sons.get(0))).intValue();
                  Nodo onlySon = (Nodo) newFP3.n.get(ind);
                  SP_FP3.addElement(onlySon.Name);
                  SP_freq.addElement(""+onlySon.freq);
                  dad++;
            }
            for (int k=0; k<SP_FP3.size(); k++){
                  int dim = KSPath.size();
                  String TMPs = (String) SP_FP3.get(k);
                  for (Enumeration e = KSPath.keys();
                         e.hasMoreElements(); ){
                        String key = (String) e.nextElement();
                        String newKey = TMPs+","+key;
                        KSPath.put(newKey, SP_freq.get(k));
                        allKPath.put(ITEM+","+newKey, ""+freqItem);
                  }
                  allKPath.put(ITEM+","+TMPs, ""+freqItem);
            }
            for (int k=0; k<SP_FP3.size(); k++){
                  String TMPs = (String) SP_FP3.get(k);
                  KSPath.put(TMPs, SP_freq.get(k));
            }
            // MULTI PATH FP_TREE
            FP_Tree MP_FP3 = new FP_Tree();
            if ((newFP3.n.size()-dad-1) >0){
                  MP_FP3.CloneFP3(newFP3, dad);

            //    chiamata ricorsiva di FP_Growth() su MP_FP3
                  KMPath = (Hashtable) FP_Growth(MP_FP3);
                  numRic--;
                  for (Enumeration e = KMPath.keys();
                                        e.hasMoreElements(); ){
                        String key = (String) e.nextElement();
```

```
                        int supp = (Integer.valueOf((String)
                                    KMPath.get(key))).intValue();
                        if ( supp > minSup){
                              String newKey = ITEM +","+ key;
                              allKPath.put(newKey, ""+freqItem);
                              for (Enumeration ee = KSPath.keys();
                                          ee.hasMoreElements(); ){
                                    String key2 = (String)ee.nextElement();
                                    String newKey2 = newKey+","+key2;
                                    allKPath.put(newKey2, ""+supp);
                              }
                        }
                  }
                  // CANCELLAZIONE ITEMS INCOMPATIBILI
                  for (Enumeration e = allKPath.keys();
                              e.hasMoreElements(); ){
                        String key = (String) e.nextElement();
                        int supp = (Integer.valueOf((String)
                                    allKPath.get(key))).intValue();
                        if (Incompatible(key))
                              allKPath.remove(key);
                  }
            }

      }
      if (numRic == 1)  return KP;
            else return allKPath; }
```

Implementazione del metodo FP_Growth

Capitolo 6
Sperimentazione del metodo

Al fine di verificare la bontà del metodo proposto e la sua robustezza anche in casi complessi, lo si è sperimentato applicandolo ad un caso concreto.

In questo capitolo, dopo aver definito i tratti salienti di un caso concreto, individuando le classi spaziali coinvolte e proponendo una query su di essi, si espongono i risultati ottenuti sotto forma di k-predicati spaziali. Tali predicati, in seguito, vengono utilizzati per approfondire meglio le fasi salienti del metodo proposto; in particolare, si analizza un k-pattern frequente mostrando, nel dettaglio, il comportamento dell'algoritmo di visita del Frequent Pattern Tree costruito sulla query spaziale.

Infine, si interpretano i dati ottenuti generando l'insieme di regole associative spaziali relative ad uno dei k-pattern risultanti, e individuando tra esse quelle maggiormente significative.

6.1 Definizione del caso

Ipotizzando uno studio sulle condizioni di vita negli USA si sono utilizzate quattordici differenti mappe relative ai seguenti argomenti:

1. Contee	**2. Fattorie**
3. Città	**4. Forza Lavoro**
5. Strade	**6. Mortalità**
7. Ferrovie	**8. Piovosità**
9. Aereoporti	**10. Criminalità**
11. Bacini idrici	**12. Reddito**
13. Dighe	

Layers analizzati

Come si può notare ad un primo esame, le mappe scelte per questa ricerca, rappresentano un'ampia varietà di classi di dati spaziali spesso senza nessuna correlazione tra loro. Il motivo che ha guidato tale scelta, risiede nella intenzione di determinare oltre a relazioni tra classi di oggetti affini, quali ad esempio il *Reddito* e la *Forza lavoro*, anche altre in grado di esprimere possibili connessioni esistenti tra classi insospettabilmente collegate, quali ad esempio *Autostrade* e il numero di *Fattorie* presenti in un territorio.

La query spaziale presa in esame è stata la seguente:
"Selezionare tutte le regole associative spaziali esistenti tra le contee e le città, le ferrovie, le grandi strade, gli aereoporti, i bacini idrici, il tasso di piovosità annuo, le dighe, il numero di fattorie, il tasso di disoccupazione, di criminalità e di mortalità riscontrati e il reddito pro capite della popolazione. Inoltre, queste regole associative spaziali devono avere un supporto minimo del 60% e una confidenza del 80%."

La prima problematica da affrontare è stata la scelta dell'attributo rappresentativo di ogni singola classe di oggetti e l'eventuale raggrupamento dei valori, relativi a tale attributo, in insiemi più generali e significativi.

Nome Classe	**Nome attributo**	Valori possibili	
Piovosità	**Tasso**	*< 30 mm*	
		30 .. 100 mm	
		> 100 mm	
Bacini Idrici	**Tipo**	*Lake*	*Bay or Ocean*
		Swamp	*Reservoir*
		Stream	*Canal*
Città	**Numero Abitanti**	*0 .. 10,000*	*10,000 .. 100,000*
		100,000 .. 500,000	*500,000 .. 1,000,000*
		>1,000,000	
Lavoro	**Tasso Disoccupazione**	*<10 %*	*10% .. 20%*
		> 20%	
Reddito	**Reddito Pro Capite Annuo**	*<70,000$*	*150,000$ . .500,000$*
		70,000 .. 150,000$	*>500,000$*

Fattorie	**Numero Fattorie**	*<1,000*	*1,000 .. 2,000*
		> 2,000	
Strade	**Tipo**	*Principal Highway*	*Ltd Acces HW*
		Other Highway	*Ferry Crossing*
Mortalità	**Numero Morti**	*<3,000*	*3,000 .. 10,000*
		>10,000	
Criminalità	**Numero di Crimini**	*<1,000*	*3,000 . .5,000*
		1,000 .. 3,000	*>5,000*
Aereoporti	**Tipo**	*Airport*	*Seaplane Base*
		Heliport	
Dighe	**Tipo**	*Dams*	
Ferrovie	**Tipo**	*Railroad*	*Railroad in tunnel*

Tabella dei valori scelti per ogni attributo

6.2 Risultati ottenuti

Applicando la query spaziale alle mappe relative agli oggetti, appartenenti alle classi spaziali scelte, presenti su tutto il territorio degli Stati Uniti d'America (compresi i territori dell'Alaska e delle Hawaii), e strutturando i dati in base alle classificazioni scelte, si sono ottenute 1877 transazioni (una per ogni contea) e 54 differenti coppie <*relazione*, *oggetto*>; Di queste, solo 21, contando le ripetizioni frequenti, sono presenti in almeno il 60% delle transazioni totali, presentano cioè una frequenza superiore a 1126 (vincolo del minimo supporto richiesto).

Se escludiamo le ripetizioni il numero di items frequenti si riduce a 9.

freq	codice	Nome:
1809	**46#1**	(crosses, road:"Other Highway")#1
1770	**7#1**	(contains, cities:"0 - 10,000")#1
1762	**47#1**	(crosses, road:"Principal Highway")#1
1750	**21#1**	(contains, labour:"<10%")#1
1728	**42#1**	(crosses, railroad:"Railroad")#1
1685	**46#2**	(crosses, road:"Other Highway")#2
1675	**16#1**	(contains, agcens:"<1000")#1
1672	**47#2**	(crosses, road:"Principal Highway")#2
1622	**7#2**	(contains, cities:"0 - 10,000")#2
1477	**46#3**	(crosses, road:"Other Highway")#3
1473	**42#2**	(crosses, railroad:"Railroad")#2
1462	**7#3**	(contains, cities:"0 - 10,000")#3
1399	**15#1**	(contains, dams:"DAM")#1
1376	**47#3**	(crosses, road:"Principal Highway")#3
1308	**7#4**	(contains, cities:"0 - 10,000")#4
1281	**12#1**	(contains, crimes:"<1000")#1
1248	**42#3**	(crosses, railroad:"Railroad")#3
1246	**46#4**	(crosses, road:"Other Highway")#4
1176	**7#5**	(contains, cities:"0 - 10,000")#5
1171	**6#1**	(contains, bea:"> 500000$")#1
1149	**47#4**	(crosses, road:"Principal Highway")#4

Tabella degli items frequenti con ripetizioni

Ottenuto l'elenco degli items frequenti passiamo ad ordinare la tabella delle transazioni, come spiegato nei capitoli 4 e 5, in modo da predisporre i dati ad essere inseriti in un FP-Tree.

L'albero così generato, possiede 21 valori nello *Header* (uno per ogni items) e un totale di 2256 nodi relativi agli items frequenti presenti nelle transazioni. Visitando, come proposto in [HP04], tale albero si sono ottenuti 684 k-patterns frequenti, così distribuiti:

k-Pattern	Totale
1-Pattern	21
2-Pattern	113
3-Pattern	235
4-Pattern	227
5-Pattern	80
6-Pattern	8
k-Pattern	*684*

Elenco dei k-pattern generati

Per determinare i patterns frequenti è necessario, come mostrato nei capitoli precedenti, eseguire una opportuna visita del FP-Tree creato.

La scelta di una opportuna forma di visita, è fondamentale per generare l'insieme dei patterns in tempi accettabili. Una prima idea potrebbe essere quella di partire dalle foglie dell'albero e risalirlo generando per ogni nuovo nodo incontrato, l'insieme dei k-patterns formati dai nodi compresi tra la foglia da cui si è partiti e il nodo in esame. Se un pattern è già stato individuato, si incrementa il suo supporto; al termine della visita, si scartano tutti i pattern aventi un supporto inferiore al minimo richiesto.

Un'altra possibilità un po' più raffinata, prevede di effettuare una visita per ogni item presente nel header e generare solo i k-pattern relativi a tale item (ovvero che hanno come primo elemento l'item in questione). In questo modo non si sarà costretti a generare preventivamente tutti i possibili k-patterns, il che richiede una notevole mole di spazio di memoria, ma solo una porzione più ristretta di essi.

Infine, una soluzione ulteriore che sfrutta al meglio la struttura e le caratteristiche di un FP-Tree, raffina la precedente iterando il principio di separazione della ricerca dei k-patterns per items diversi.

Tale visita prevede, per ogni item *i* presente nel header, di analizzare il sottoalbero formato dai soli nodi attraversati da un cammino che dai nodi relativi all'item in esame risale fino alla radice. Un albero così fatto è detto Conditional FP-Tree (*CFP-Tree*) dello

item *i* e rappresenta l'insieme delle sole transazioni (ordinate) che contengono tale item, troncate eliminando tutti gli items aventi frequenza inferiore a quella di *i*.

Il vantaggio di creare un Conditional FP-Tree risiede nella possibilità di ridurre enormemente la dimensione dell'albero da analizzare, riducendo perciò anche i tempi di elaborazione. Inoltre, tale strategia può essere riapplicata ricorsivamente generando, per ogni item presente nello header del conditional FP-Tree, il proprio CFP-Tree, ottenendo via via insiemi di nodi sempre più ristretti fino a giungere ad un albero vuoto.

L'efficacia del Conditional FP-Tree risiede nel fatto che tutti i nodi che vengono scartati al momento della sua costruzione, sono relativi a transazioni che non contengono l'item di cui si stanno generando i k-pattern; quindi non devono essere presi in considerazione. Invece, quelli presenti nell'albero appena creato, sono relativi ad items che, potenzialmente, si trovano "spesso" nelle stesse transazioni in cui si trova l'item *i*. Questo vantaggio è ancora più marcato al momento di creare il Conditional FP-Tree di un item *k* da quello di *i*; infatti, in tale costruzione, non generando i nodi superflui, si impedisce di verificare la validità di k-pattern della forma *{i, k, j}* in cui l'item *j* non è frequentemente individuabile nelle transazioni in cui appaiono contemporaneamente i nodi *i* e *k*. Ovviamente, questo non impedisce di avere pattern frequenti aventi al loro interno le coppie (*i, j*) o (*k, j*).

Osservazione 1:

Il Conditional FP-Tree di un item *i* non genera tutti i k-patterns frequenti che lo contengono (cosa che per altro comporterebbe la generazione in *j* situazioni diverse dello stesso *j*-pattern), ma soltanto l'insieme dei k-patterns aventi *i* come primo item.

Osservazione 2:

Il Conditional FP-Tree di un nodo *i* accetta come validi solo i nodi che si trovano a livelli superiori di quelli dello item in esame; ciò comporta che gli items proprietari avranno certamente un supporto maggiore. Poiché per generare un k-pattern genero il Conditional FP-Tree relativo ad items presenti, si può affermare che qualunque k-patterns ha gli items ordinati per frequenza crescente:

$$Supporto(\text{Item}_1) \leq Supporto(\text{Item}_2) \leq \ldots \leq Supporto\ (\text{Item}_k)$$

Inoltre il suo supporto sarà dato da:

$$\text{Supporto}(C.FP\text{-}Tree(\text{Item}_1) \cap C.FP\text{-}Tree(\text{Item}_2) \cap \ldots \cap C.FP\text{-}Tree(\text{Item}_k))$$

Per meglio comprendere i passi che, dalla costruzione del FP-Tree, ci hanno portato all'individuazione di tali patterns, utilizziamo, come campione esemplificativo, uno dei k-patterns generati.

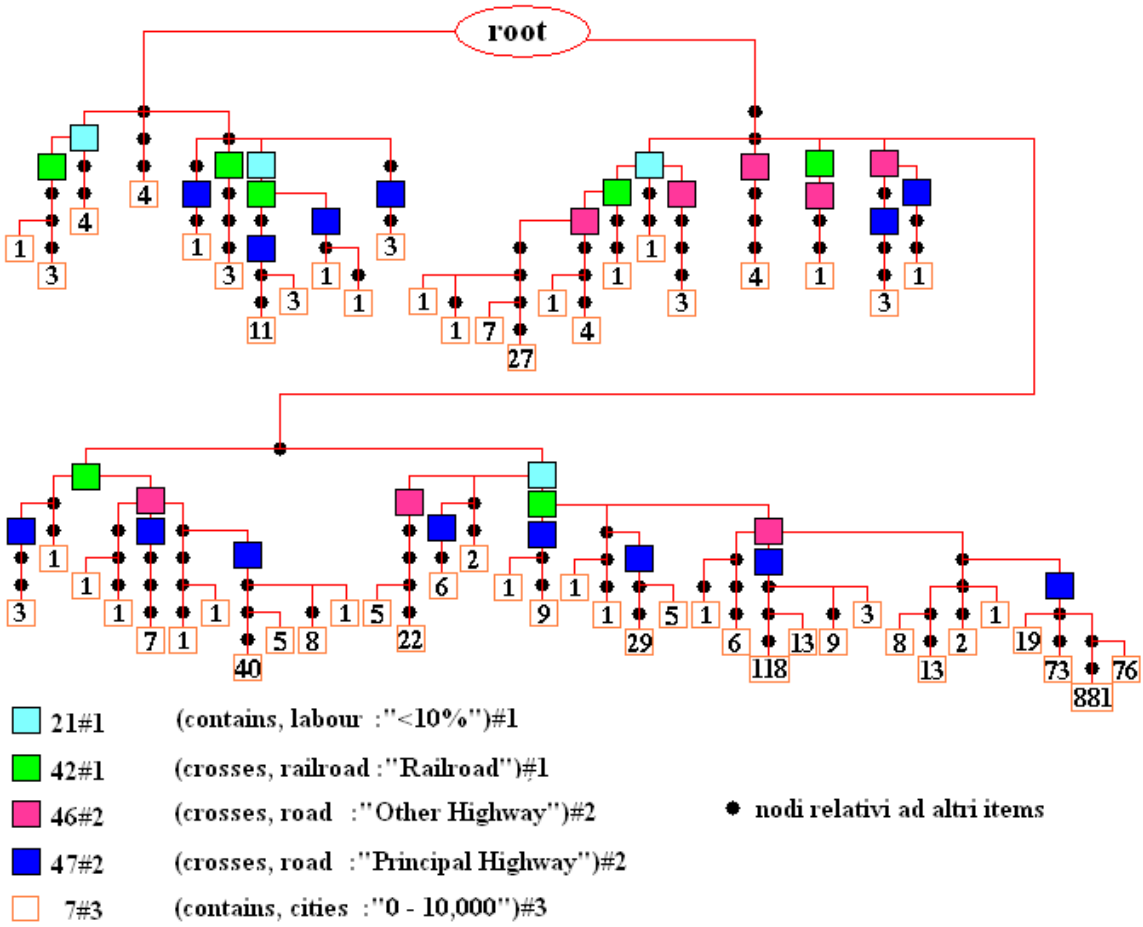
{(contains,cities:"0-10,000")#3, (crosses,roads:"Principal Highway")#2, (crosses,roads:"Other Highway")#2, (crosses,railroads:"Railroad")#1, (contains,labour:"<10%")#1}

Tale 5-Pattern ha un supporto di 1253, pari al 66.7% del totale. Tale valore può essere rilevato contando il numero di transazioni che possiedono tutti e 5 gli items che compongono il pattern.

Codice	Nome item	Frequenza
7#3	*(contains,cities:"0-10,000")#3*	**1462**
47#2	*(crosses, roads:"Principal Highway ")#2*	**1672**
46#2	*(crosses,roads:"Other Highway")#2*	**1685**
42#1	*(crosses,railroads:"Railroad")#1*	**1728**
21#1	*(contains,labour:"<10%")#1*	**1750**

Tabella degli items del 5-pattern

Come precedentemente anticipato, tale k-pattern può essere generato esclusivamente dal Conditional FP-Tree del primo item, ovvero da *(contains,cities:"0-10,000")#3*.



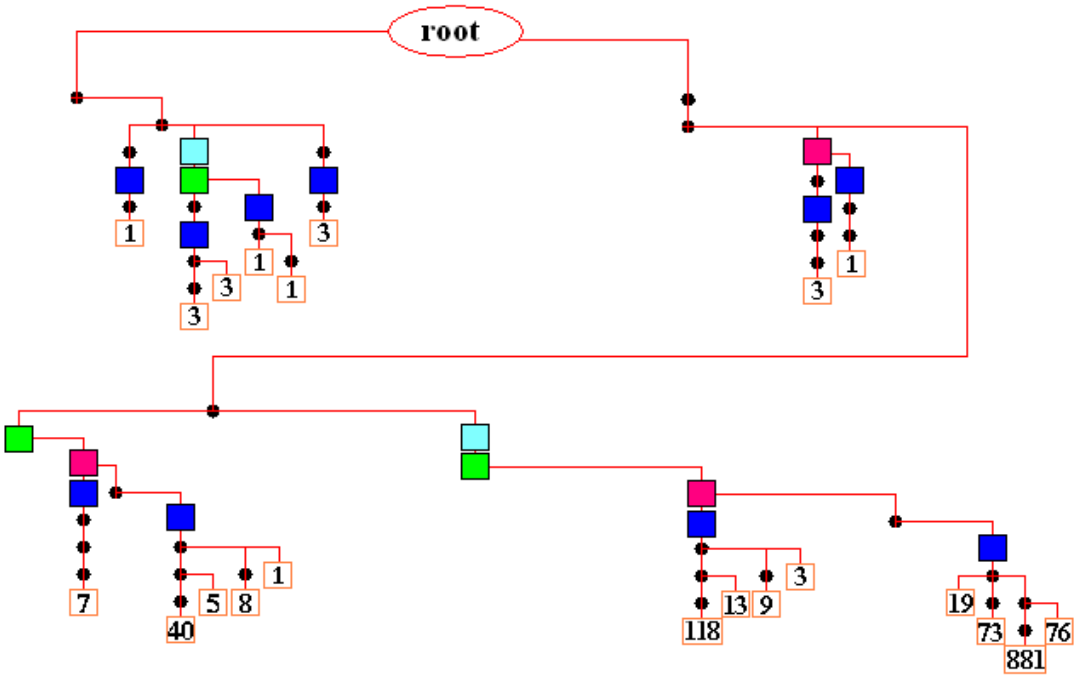
conditional-FP-Tree dell'item 7#3

In figura, si evidenziano i nodi relativi agli items facenti parte del pattern d'esempio; inoltre, i valori presenti nei riquadri relativi all'item con codice *7#3*, indicano la frequenza relativa di ogni singolo nodo foglia la cui somma ci fornisce la frequenza assoluta dell'item in questione.

Poichè in quest'albero vi sono nodi appartenenti ad items diversi, possiamo, per ognuno di essi, generare il proprio Conditional FP-Tree; in questo modo potremo ottenere un nuovo albero, più ristretto, da cui poter estrarre k-patterns della forma:

$$\{(contains,cities:"0\text{-}10,000")\#3,\ j\ , \ldots\}$$

Poichè il secondo item che appare nel 5-pattern d'esempio è: *(crosses,roads:"Principal Highway")#2*, applichiamo la procedura di generazione del Conditional FP-Tree ai nodi di tale item che sono presenti nell'albero attuale.



primo passo di ricorsione

Come si può notare, sono stati eliminati i nodi relativi a cammini che non avevano tutti e due gli items, e il numero totale di nodi si è nettamente ridotto.

Osservazione 3:

L'albero che abbiamo generato è il conditional FP-Tree dello item *(crosses,roads:"Principal Highway")#2* applicato al conditional FP-Tree dello item *(contains,cities:"0-10,000")#3*.

Lo stesso albero può essere rappresentato mediante la formula:

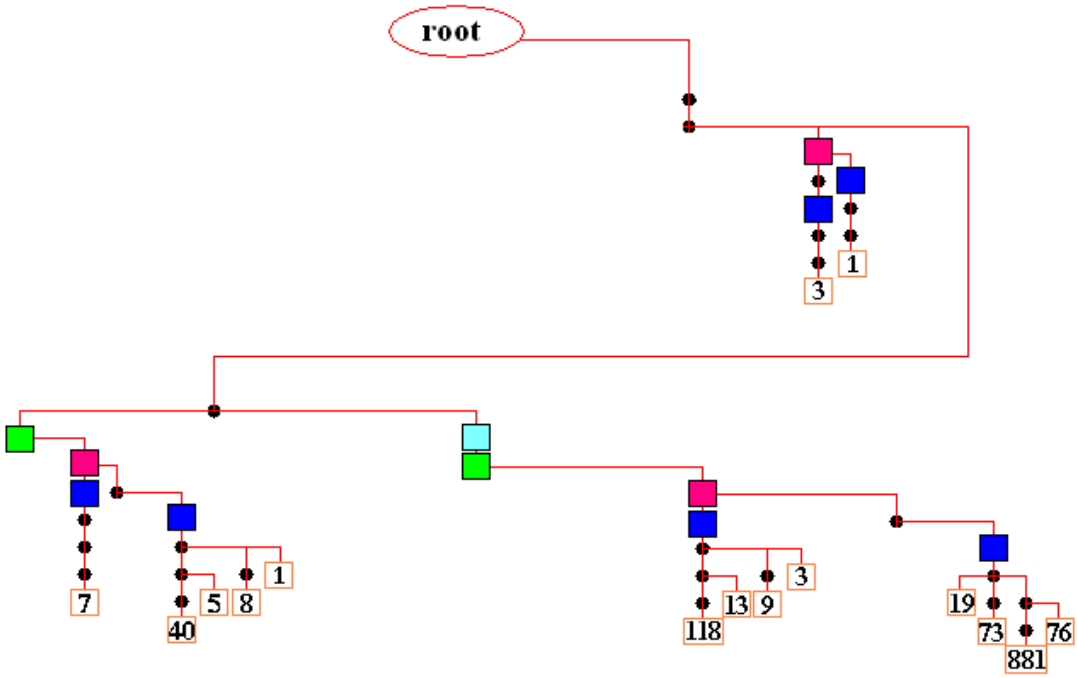
Cond.FP-Tree(Cond.FP-Tree(FP-Tree, 7#3), 47#2)

Con questa rappresentazione si mette in risalto il carattere ricorsivo del procedimento. Inoltre, tale formula è equivalente all'affermazione tra insiemi:

Cond.FP-Tree(7#3) ∩ Cond.FP-Tree(47#2)

Da quest'ultima formula, sfruttando l'osservazione 2, possiamo facilmente dedurre il supporto.

Supporto(Cond.FP-Tree(7#3) ∩ Cond.FP-Tree(47#2))



secondo passo di ricorsione

I cammini che sono rimasti, hanno al loro interno nodi relativi ai tre items componenti il pattern.

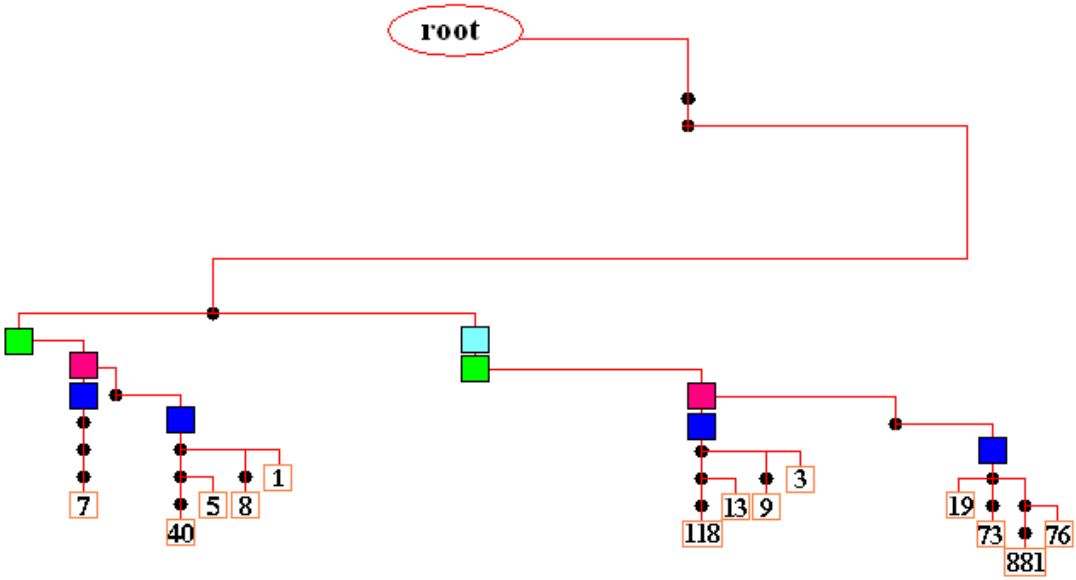
{(contains,cities:"0-10,000")#3, (crosses,roads:"Principal Highway")#2,

(crosses,roads:"Other Highway")#2}.

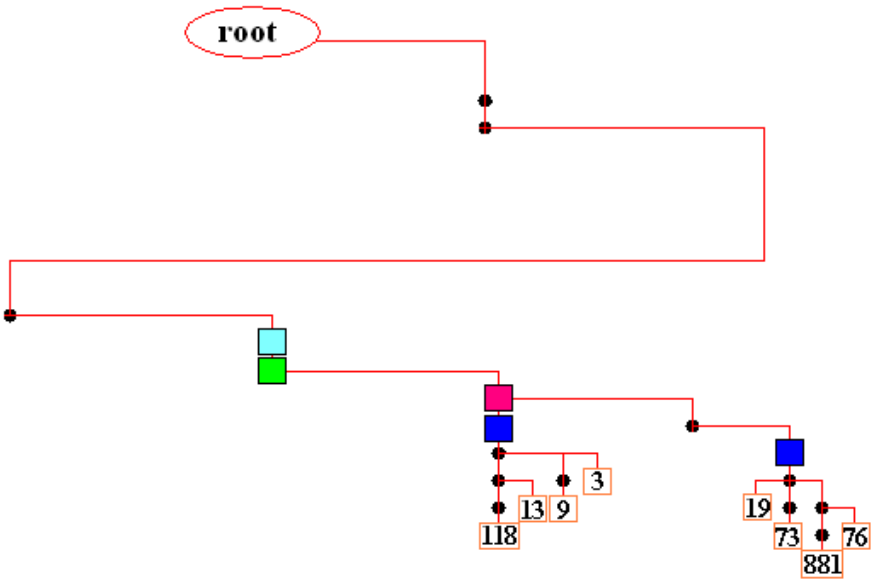
Ripetendo il procedimento per i restanti items del 5-pattern d'esempio, vediamo che l'albero rapidamente si spoglia dei rami non validi fino a quando, giunti alla quinta ricorsione,

Cond.FP-Tree(Cond.FP-Tree(Cond.FP-Tree(Cond.FP-Tree(Cond.FP-Tree(FP-Tree, 7#3), 47#2), 46#2), 42#1), 21#1)

Otteniamo un albero in cui ogni cammino contiene tutti e 5 gli items.



terzo passo di ricorsione



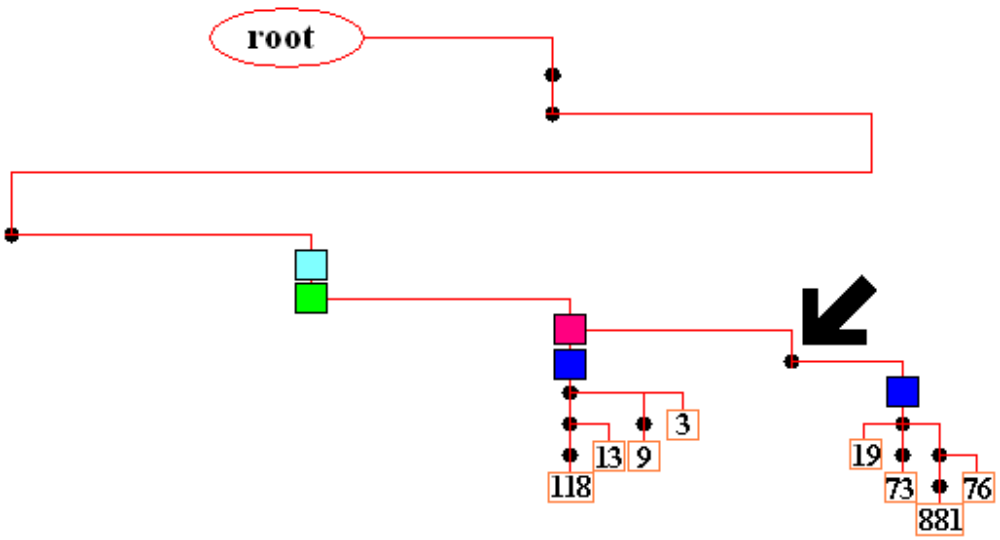
quarto passo di ricorsione

La generazione del conditional FP-Tree relativo al 5-pattern dell'esempio ci mostra tutti e soli i cammini possibili che dalla radice giungono ad una foglia attraversando i nodi di tutti gli items in questione. Sommando semplicemente le frequenze relative ai nodi foglie otterremo il supporto relativo al 5-pattern in esame.

Analizzando l'albero finale che si è generato, possiamo notare la presenza di altri items che potenzialmente potremmo aggiungere al 5-pattern ottenuto creandone così degli altri di lunghezza maggiore. In particolare, i tre nodi che precedono, nell'unico cammino rimasto, i nodi del 5-pattern se aggiunti potrebbero esprimere relazioni più complesse di quelle definibili con l'esempio fatto finora. Purtroppo però, i tre nodi sono relativi ai seguenti items:

- *(crosses,roads:"Other Highway")#1* il primo,
- *(contains,cities:"0-10,000")#1* il secondo e
- *(crosses,roads:"Principal Highway")#1* il terzo nodo

che sono ripetizioni diverse di items già presenti nel k-pattern.
Gli altri nodi rappresentati sono quasi tutti ripetizioni degli items analizzati e quindi inutili. L'unico nodo a essere interessante è quello indicato in figura dalla freccia. Purtroppo però, il supporto che il nuovo 6-pattern avrebbe è di 1049, inferiore al minimo richiesto, quindi infrequente e da scartare. In conclusione possiamo affermare che non vi sono k-pattern più lunghi di 5 che contengono al loro interno tutti gli items che abbiamo scelto.



possibile item da aggiungere

6.3 Interpretazione dei k-pattern ottenuti

Nel paragrafo precedente si è mostrato come, analizzando il Frequent Pattern Tree sia possibile giungere alla determinazione dei pattern frequenti. Una volta trovati però, è necessario raffinarli effettuando per ognuno di essi analisi più approfondite. Uno dei modi più noti per dare significato agli itemsets è quello di generare regole associative; nel nostro caso, poichè trattiamo dati con valenza geografica, si parla di regole associative spaziali.

Il concetto secondo cui tutti gli items, e conseguentemente tutti i patterns finora calcolati sono relazioni tra gli oggetti della classe referente (le contee nel nostro esempio) e quelli presenti nei vari layer è rimasto finora latente. Per esplicitarlo introduciamo il nuovo item *is_a(x, Country)* che esprime che un oggetto *x* è una contea e lo includiamo in ogni k-pattern finora generato.

Prendiamo come esempio il 4-pattern esteso:
{ is_a(x, Country) , (contains,crimes:"<1000")#1, (contains,labour:"<10%")#1, (contains, cities:"0 - 10,000")#1}
ed elenchiamo tutte le possibili regole in esso contenute, ricordando che una regola associativa è una implicazione tra predicati (nel nostro caso spaziali) della forma ***A*** → ***B*** **(s, c)**, in cui:

- ***A*** e ***B*** sono predicati costruiti come congiunzioni di items spaziali cioè:
  $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m)$.
- ***s*** è il **supporto** cioè la probabilità che i predicati *A* e *B* appaiano insieme nella stessa transazione $supporto(A, B) = Prob\{ A \cup B \}$.
- ***c*** è la **confidenza**, cioè la probabilità che B si verifichi se A si è verificato.

Per semplicità di trattazione abbreviamo i 4 items presenti nel predicato.

is_a(x, County)	A
(contains,crimes:"<1000")#1	B
(contains,labour:"<10%")#1	C
(contains, cities:"0 - 10,000")#1	D

Tabella delle abbreviazioni usate

Combinando opportunamente gli items e ricordando la formula che lega la confidenza di una regola al supporto dei suoi predicati:

$$\text{confidenza}(A \rightarrow B) = \text{Prob}\{B|A\} = \frac{\text{Prob}\{A \cup B\}}{\text{Prob}\{A\}} = \frac{\text{supporto}(A, B)}{\text{supporto}(A)}$$

possiamo calcolare tutte le possibili regole associative ed la loro confidenza:

Regole Associative spaziali	Confidenza	
{A} → {B,C,D}	1142/1877	60.8%
{B} → {A,C,D}	**1142/1281**	**89.1%**
{C} → {A,B,D}	1142/1750	65.2%
{D} → {A,B,C}	1142/1770	64.5%
{A,B} → {C,D}	**1142/1281**	**89.1%**
{A,C} → {B,D}	1142/1750	65.2%
{A,D} → {B,C}	1142/1770	64.5%
{B,C} → {A,D}	**1142/1189**	**96.0%**
{B,D} → {A,C}	**1142/1230**	**92.8%**
{C,D} → {A,B}	1142/1649	69.3%
{A,B,C} → {D}	**1142/1189**	**96.0%**
{A,B,D} → {C}	**1142/1230**	**92.8%**
{A,C,D} → {B}	1142/1649	69.3%
{B,C,D} → {A}	**1142/1142**	**100%**

Tabella delle regole associative spaziali

Per calcolare la confidenza di una regola associativa, è necessario conoscere il supporto di tutti i j-pattern (con $1 \leq j < k$) relativi agli stessi items. Poichè per la proprietà degli itemsets secondo cui se un k-itemset ha frequenza *q*, qualunque n-itemset (con $n \geq k$) che lo contiene avrà una frequenza *p* minore o al più uguale a *q* ($p \leq q$), allora tutti j-pattern necessari sono anch'essi frequenti e quindi la loro frequenza ci è nota.

Tutte le regole che hanno passato il test del vincolo di minima confidenza, vengono definite "*strong rules*" in quanto esprimono la relazione che se una transazione del nostro database contiene gli items a sinistra dell'implicazione, è molto probabile che possa contenere anche quelli a destra.

E' interessante notare che cambiando la direzione dell'implicazione si hanno dei risultati completamenti diversi. E' il caso, ad esempio, della regola {A,B,C} → {D} che ha una confidenza molto alta (la più alta se escludiamo la regola banale {B,C,D} → {A}), pari al 96.0% e che esprime il concetto:

"*Se una contea ha un basso numero di crimini commessi (minore di 1000) e di disoccupazione (inferiore al 10%) allora possiede una città con pochi abitanti (meno di 10.000)*".

Invece, La regola {A,D} → {B,C} che ci dice che:

"*Se una contea possiede una piccola cittadina allora si riscontrerà una bassa criminalità e un basso tasso di disoccupazione*", ha una confidenza, come ci si potrebbe aspettare, insufficiente (pari al 64.5%).

Conclusioni

Il principale scopo che ha guidato la realizzazione e la stesura di questa tesi è stato quello di realizzare e presentare un metodo di integrazione tra il processo di knowledge discovery tradizionale e il modello di rappresentazione dei dati spaziali usato nei GIS al fine di determinare regole associative spaziali in grado di esprimere le correlazioni esistenti tra classi di oggetti spaziali.

Nella prima parte, dopo aver fatto il punto sull'attuale stato dell'arte introducendo alcuni tra i metodi più noti in letteratura per analizzare i dati spaziali, si sono definiti i concetti generali di GIS, con particolare attenzione ai metodi più noti di indicizzazione dello spazio rappresentato, e di Knowledge Discovery, in cui dopo aver evidenziato le fasi salienti di tale processo, si sono discussi, mettendoli a confronto, l'algoritmo classico di determinazione degli itemsets frequenti, l'Apriori, e una soluzione alternativa che, sfruttando un ordinamento preventivo dei dati presenti nelle transazioni, costruisce prima e visita poi, un albero detto Frequent Pattern Tree.

Nella seconda parte di questa tesi, si è presentata una nuova metodologia per ricercare conoscenza tra i dati spaziali presenti in un GIS. Tale soluzione prende spunto dal procedimento presentato da Kopersky e Han in **[KH95]**, in cui, attraverso una serie di fasi di generalizzazione, sfruttando le predefinite gerarchie sui predicati spaziali e sulle classi, si passa dai dati "grezzi" raccolti in un database transazionale ad items via via più interessanti per generare poi, attraverso una metodologia che ricalca la filosofia di Apriori, i k-patterns frequenti.

A differenza di questa soluzione però, in questo scritto si affronta il problema della determinazione degli stessi k-patterns anticipando la fase di generalizzazione dei dati al momento stesso della raccolta e definendo una struttura ad albero per meglio trattare gli

items interessanti; in questo modo, pur perdendo la possibilità di analizzare i dati a diversi livelli di dettaglio, si snellisce la fase preliminare rendendo più agevole l'intero processo.

In base a questo principio, al termine di questa prima fase di estrazione delle informazioni grezze, si avrà un database transazionale contenente tutte le informazioni che ci serviranno nella definizione delle regole associative spaziali.

Al fine di trattare in maniera efficiente questo database che può essere di grandi dimensioni, si costruisce su di esso un Frequent Pattern Tree; in questo modo, con una semplice visita (per determinare quali items sono frequenti) e una fase di ordinamento delle transazioni si riesce a organizzare i dati per inserirli nell'albero. Oltre al fatto che tale albero permette di ridurre notevolmente la dimensione dei dati da analizzare, un FP-Tree è in grado di trovare gli itemsets, detti anche k-patterns, frequenti in maniera "furba", non generando, come fa l'Apriori, l'insieme dei candidati, da dover poi selezionare al fine di scegliere i valori realmente validi.

Caratteristica peculiaria di questo lavoro, che lo differenzia dagli altri presenti in letteratura, risiede nel trattamento delle repliche dei dati, non più viste semplicemente come ridondanze inutili di informazioni, ma come un'ulteriore elemento da cui apprendere conoscenza. Infatti, più volte all'interno di questo scritto si è mostrato come l'utilizzo di un item che rappresenta una coppia <*relazione,classse*> ripetuta *n* volte sia in grado di esprimere relazioni più forti di quelle che un items "semplice" è in grado di comunicare.

Ovviamente, questo surplus di conoscenza per essere modellato necessita di particolari accortezze, al fine di impedire la definizione di patterns, e conseguentemente di regole associative, inconsistenti ed inutili. Per questo motivo, al momento della valutazione di un pattern per determinare se sia frequente o meno, si effettua anche una verifica sulla sua validità, verificando se vi siano due o più items relativi alla stessa coppia spaziale cancellando così tutti quei patterns che esprimono relazioni tra ripetizioni diverse di uno stesso items.

La fase realizzativa vera e propria ha portato all'implementazione di due distinte componenti: la prima costituita da un plug-in del GIS JUMP, in grado di rilevare tutte le

informazioni utili presenti nei layers inseriti nel sistema, la seconda relativa alla realizzazione di un sistema in grado di generare l'insieme dei k-patterns da cui poter generare tutte le regole associative spaziali.

Il metodo è stato, infine, sperimentato su un caso di studio concreto per verificarne la validità e la sua efficacia ed ottenendo risultati interessanti.

In futuro, sarebbe interessante sviluppare il metodo proposto in questa tesi effettuando una valutazione prestazionale al fine di individuare eventuali colli di bottiglia da risolvere. Inoltre, è auspicabile una maggiore flessibilità del metodo al fine di permettere all'utente di intervenire attivamente nelle varie fasi del processo, ottenendo così, risultati più interessanti per i suoi scopi. Infine, sarebbe utile ampliare la discussione sulle gerarchie (delle relazioni e sulle classi) proposte e sulla loro eventuale estndibilità.

Per concludere, questa tesi può essere vista come un punto di partenza per realizzare un completo sistema in grado di effettuare ricerche ed analisi più approfondite (ad esempio estendendo il metodo ai valori spazio-temporali) sui dati contenuti in un GIS.

Appendice: codice commentato

Per la creazione di questa appendice si è utilizzato *Doxygen ver. 1.4.1*, strumento utile nella generazione di automatica di documentazione per codice C, C++, Java, Objective-C, IDL (Corba and Microsoft flavors) alcune estensioni di PHP, C#, e D. **[DOXY]**

Sommario

KDDonGIS: Directory

Questa gerarchia di directory è ordinata in ordine alfabetico:

KDDonGIS: Lista dei package

Questi sono i package e una loro breve descrizione:

KDDonGIS: Lista dei composti

Queste sono le classi, structs, unions e interfacce con una loro breve descrizione:

KDDonGIS: Lista dei file

Questa è una lista di tutti i file con una loro breve descrizione:

KDDonGIS Documentazione delle directory

Riferimenti per la directory KDonGIS/Directory

- directory parsing
- directory plugins

Riferimenti per la directory KDonGIS/parsing

- file FP_Tree.java
- file Head.java
- file Item.java
- file Nodo.java
- file ParseTransaction.java

Riferimenti per la directory KDonGIS/plugins

- file SpatialTransactionsGeneratorExtension.java
- file SpatialTransactionsGeneratorPlugIn.java

KDDonGIS Documentazione dei namespace

Package parsing

parsing

Composti

- class FP_Tree
- class Head
- class Item
- class Nodo
- class ParseTransaction

Package plugins

plugins

Composti

- class SpatialTransactionsGeneratorExtension
- class SpatialTransactionsGeneratorPlugIn

KDDonGIS Documentazione delle classi

classe parsing.FP_Tree

Membri pubblici

- void CloneFP3 (FP_Tree f, int d)
- void insertTransaction (Vector v, int f)
- int searchAmongSons (int pp, String s)
- void addNewSon (int padre, String v)
- void loadHeader (String n, int f)
- void Copy (Nodo nodo)
- void print ()
- void printHeader ()
- void clear ()

Attributi pubblici

- Vector n
- Vector h

Funzioni con visibilità di package

- FP_Tree ()

Descrizione Dettagliata

Implementazione di un Frequent Pattern tree.

Autore:

Luigi Scrimitore (scrimito@cli.di.unipi.it)

Documentazione dei costruttori e dei distruttori

parsing.FP_Tree.FP_Tree () [package]

Costruttore di FP_Tree.

```
Definizione alla linea 21 del file FP_Tree.java.21          {
22              n = new Vector();
23              n.addElement(new Nodo("root"));
24              h = new Vector();
25      }
```

Documentazione delle funzioni membro:

void parsing.FP_Tree.addNewSon (int *padre*, String *v*)

Aggiunge un nuovo nodo al Frequent Pattern Tree.

Parametri:

int *Padre*, *nodo padre*
int *V* *nome nodo*

```
Definizione alla linea 86 del file FP_Tree.java.86                    {
87          int pos = n.size();
88          // aggiornamento del Header File
89          int nb = 0;
90          for (int i=0; i<h.size(); i++){
91              if ((((Head) h.get(i)).NameItem).compareTo(v) == 0){
92                  nb = ((Head) h.get(i)).firstPosition;
93                  ((Head) h.get(i)).firstPosition = pos;
94              }
95          }
96
97          n.addElement(new Nodo(v, padre, nb));
98          ((Nodo) n.get(padre)).sons.addElement(""+pos);
99      }
```

---

void parsing.FP_Tree.clear ()

Cancella tutti i nodi del Frequent Pattern Tree.

```
Definizione alla linea 140 del file FP_Tree.java.140          {
141         n.clear();
142         h.clear();
143     }
```

---

void parsing.FP_Tree.CloneFP3 (FP_Tree *f*, int *d*)

Aggiorna il FP_tree inserendo tutti i nodi di f discendenti da d.

Parametri:

FP_Tree *f*, *FP_Tree da clonare*
int *d* *nodo da cui iniziare la duplicazione*

```
Definizione alla linea 27 del file FP_Tree.java.27                    {
28          n = (Vector) f.n.clone();
29          if (d >0){
30              for (int i=1; i<=d; i++)
31                      n.remove(1);
32
33              for (int i=1; i<n.size(); i++){
34                      ((Nodo) n.get(i)).nextBrother -=d;
35                      ((Nodo) n.get(i)).dad -= d;
36                  Vector sons = (Vector)((Nodo) n.get(i)).sons;
```

```
37                          for (int j=0; j<sons.size(); j++){
38                              int pos = Integer.valueOf((String) sons.get(j)).intValue() - d;
39                              ((Vector)((Nodo) n.get(i)).sons).remove(j);
40                              ((Vector)((Nodo) n.get(i)).sons).insertElementAt(""+pos,j);
41                          }
42                      }
43                      int tot = f.h.size();
44                      for (int i=0; i<f.h.size(); i++){
45                          if (((Head) f.h.get(i)).firstPosition >d){
46                              h.addElement((Head) f.h.get(i));
47                              ((Head) h.lastElement()).firstPosition -=d;
48                          }
49                      }
50
51                      ((Nodo) n.get(0)).sons = (Vector)((Nodo) f.n.get(d)).sons.clone();
52              } else h = (Vector) f.h.clone();
53
54      }
```

---

void parsing.FP_Tree.insertTransaction (Vector *v*, int *f*)

Inserisce una nuova transazione nell'albero

Parametri:

Vector *v*,	*vettore di items*
int f	*frequenza della transazione*

```
Definizione alla linea 56 del file FP_Tree.java.56                                  {
57 //          System.out.println("TRANSAZIONE ....."+v);
58              int ptr = 0;
59              ((Nodo) n.get(ptr)).freq += f;
60
61              for (int i=0; i<v.size(); i++){
62                  int intTMP = searchAmongSons(ptr, (String) v.get(i));
63                  if (intTMP > -1){
64                      ptr = intTMP;
65                  } else {
66                      // nuovo figlio
67                      addNewSon(ptr, (String) v.get(i));
68                      ptr = n.size()-1;
69                  }
70                  ((Nodo) n.get(ptr)).freq +=f;
71              }
72      }
```

---

void parsing.FP_Tree.loadHeader (String *n*, int *f*)

aggiunta di un nuovo item nel Header del Frequent Pattern Tree.

Parametri:

String *n*, *nome item*
int *f*, *frequenza item*

```
Definizione alla linea 101 del file FP_Tree.java.101                    {
102             int pos = f;
103             Head head = new Head(n, pos);
104             h.addElement(head);
105     }
```

---

void parsing.FP_Tree.print ()

Visualizza elenco dei nodi del FP_Tree.

```
Definizione alla linea 112 del file FP_Tree.java.112                    {
113             System.out.println("---------------------- FP-TREE ----------------------------");
114             for (int i=0; i<n.size(); i++){
115                     Vector TMP = (Vector)((Nodo) n.get(i)).sons;
116                     System.out.print(i+"-esimo nodo == "+ ((Nodo) n.get(i)).Name+" fratello: "+ ((Nodo)
n.get(i)).nextBrother);
117                     for (int j=0; j<TMP.size(); j++){
118                             int pos = Integer.valueOf((String) TMP.get(j)).intValue();
119                             System.out.print("\t"+pos);
120                     }
121                     System.out.println();
122             }
123     }
```

---

void parsing.FP_Tree.printHeader ()

Visualizza l'Header del FP_Tree

```
Definizione alla linea 125 del file FP_Tree.java.125                    {
126             System.out.println("---------------------- Header FP-TREE ----------------------------");
127             for (int i=0; i<h.size(); i++){
128                     System.out.print(i+"-esimo nodo == "+((Head) h.get(i)).NameItem);
129                     System.out.print("\tfreq : .."+((Head) h.get(i)).freq);
130                     int ptr = ((Head) h.get(i)).firstPosition;
131                     while(ptr > 0){
132                             System.out.print("\t-->"+ptr);
133                             ptr = ((Nodo) n.get(ptr)).nextBrother;
134                     }
135                     System.out.println();
136             }
137
138     }
```

---

int parsing.FP_Tree.searchAmongSons (int *pp*, String *s*)

Verifica se l'item *s* è tra i figli del nodo *pp*

Parametri:

int pp,	*nodo padre*
String s	*nome dell'item da ricercare*

```
Definizione alla linea 74 del file FP_Tree.java.74                                    {
75              Vector vectorOfSons = ((Nodo) n.get(pp)).sons;
76              for(int i=0; i<vectorOfSons.size(); i++){
77                      int pos = Integer.valueOf((String) vectorOfSons.get(i)).intValue();
78                      if (s.compareTo(((Nodo)n.get(pos)).Name)== 0){
79                              return pos;
80                      }
81              }
82              //      figlio non trovato
83              return -1;
84      }
```

Documentazione dei dati membri

FP_Tree.java.Vector parsing.FP_Tree.h

Vettore di oggetti "Head" che costituisce lo Header del Frequent Pattern Tree.

FP_Tree.java.Vector parsing.FP_Tree.n

Vettore di oggetti "Nodo" che forma la struttura del Frequent Pattern Tree.

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/parsing/FP_Tree.java

classe parsing.Head

Funzioni con visibilità di package

- Head (String n, int f, int fp)
- Head (String n, int f)

Attributi con visibilità di package

- String NameItem
- int freq
- int firstPosition

Descrizione Dettagliata

Implementazione di un elemento del header del frequent Pattern Tree. Contiene: nome dell'item, frquenza e posizione nel FP-Tree del primo nodo della catena di nodi relativi al item in questione.

Autore:

Luigi Scrimitore (scrimito@cli.di.unipi.it)

Documentazione dei costruttori e dei distruttori

parsing.Head.Head (String *n*, int *f*, int *fp*) [package]

Costruttore generico di una riga del Header del FP-Tree.

Parametri:

String	*n*,	*nome del item*
int	*f*,	*frequenza item*
int	*fp*	*posizione primo nodo nel FP-Tree*

```
Definizione alla linea 18 del file Head.java.18                    {
19              NameItem = n;
20              freq = f;
21              firstPosition = fp;
22          }
```

---

parsing.Head.Head (String *n*, int *f*) [package]

Costruttore generico di una riga del Header del FP-Tree. Puntatore vuoto.

Parametri:

String	*n*,	*nome del item*
int	*f*,	*frequenza item*

```
Definizione alla linea 24 del file Head.java.24                    {
25                  NameItem = n;
26                  freq = f;
27                  firstPosition = -1;
28              }
```

Documentazione dei dati membri

int parsing.Head.firstPosition [package]

Punatore intero al primo elemento (nel FP-Tree) della catena di nodi fratelli.

Definizione alla linea 16 del file Head.java.int parsing.Head.freq [package]

Supporto del item relativo.

Definizione alla linea 15 del file Head.java.String parsing.Head.NameItem [package]

Nome dell'item.

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/parsing/Head.java

classe parsing.Item

Membri pubblici

- void print ()

Funzioni con visibilità di package

- Item (String n, String c, int f)

Attributi con visibilità di package

- String Nome
- String codice
- int freq

Descrizione Dettagliata

Classe che implementa un item.

Autore:

Luigi Scrimitore (scrimito@cli.di.unipi.it)

Documentazione dei costruttori e dei distruttori

Costruttore di un oggetto Item.

parsing.Item.Item (String *n*, String *c*, int *f*) [package]

Parametri:

String	*n*,	*Nome del item da creare*
String	*c*,	*Codice del item*
int	*f*	*frequenza del item*

```
Definizione alla linea 18 del file Item.java.18                    {
19              Nome = n;
20              codice = c;
21              freq = f;
22      }
```

Documentazione delle funzioni membro

void parsing.Item.print ()

Visualizzazione dei dati di un item.

```
Definizione alla linea 24 del file Item.java.24                    {
25              System.out.print("Nome: "+Nome);
26              System.out.print("\tCodice: "+codice);
27              System.out.println("\tFrequenza: "+freq);
28      }
```

Documentazione dei dati membri

String parsing.Item.codice [package]
Codice associato ad un item.

Item.java.int parsing.Item.freq [package]
Frequenza relativa di un item.

Item.java.String parsing.Item.Nome [package]
Nome del item.

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/parsing/Item.java

classe parsing.Nodo

Funzioni con visibilità di package

- Nodo ()
- Nodo (String n)
- Nodo (String n, int d, int nb)
- Nodo (int d, int f, String n)

Attributi con visibilità di package

- String Name
- int freq
- int dad
- int nextBrother
- Vector sons

Descrizione Dettagliata

Classe che implementa un nodo del Frequent Pattern Tree.

Autore:

Luigi Scrimitore (scrimito@cli.di.unipi.it)

Documentazione dei costruttori e dei distruttori

parsing.Nodo.Nodo () [package]

Costruttore di un nodo vuoto del albero.

```
Definizione alla linea 24 del file Nodo.java.24          {
25              sons = new Vector();
26      }
```

---

parsing.Nodo.Nodo (String *n*) [package]

Costruttore di un nodo del albero. E' usata solo per creare il nodo "root".

Parametri:

int *n* *nome del nodo. Corrisponde al codice del item relativo.*

```
Definizione alla linea 28 del file Nodo.java.28          {
29              // usata solo per creare il nodo root
30              Name = n;
31              dad = -1;
32              freq = 0;
33              sons = new Vector();
34              nextBrother = -1;
35      }
```

---

parsing.Nodo.Nodo (String *n*, int *d*, int *nb*) [package]

Costruttore di un nodo del albero. Si specifica il prossimo fratello.

Parametri:

int *n*,	*Nome del nodo. Corrisponde al codice del item relativo.*
int d,	*Puntatore al padre del nodo nel albero.*
int Nb	*Puntatore al prossimo fratello nella catena di nodi relativi allo stesso item.*

```
Definizione alla linea 37 del file Nodo.java.37                    {
38              Name = n;
39              dad = d;
40              freq = 0;
41              sons = new Vector();
42              nextBrother = nb;
43      }
```

---

parsing.Nodo.Nodo (int *d*, int *f*, String *n*) [package]

Costruttore di un nodo del albero. Si specifica la frequenza.

Parametri:

int d,	*Puntatore al padre del nodo nel albero.*
int f,	*Frequenza relativa del nodo*
int *n*	*Nome del nodo. Corrisponde al codice del item relativo.*

```
Definizione alla linea 46 del file Nodo.java.46                    {
47              Name = n;
48              dad = d;
49              freq = f;
50              sons = new Vector();
51              nextBrother = -1;
52      }
```

Documentazione dei dati membri

int parsing.Nodo.dad [package]

Puntatore intero al nodo padre.

Nodo.java.int parsing.Nodo.freq [package]

Supporto relativo del nodo.

Nodo.java.String parsing.Nodo.Name [package]

Nome del nodo. Corrisponde al codice del item presente nel Header del FP-Tree.

Nodo.java.int parsing.Nodo.nextBrother [package]

Puntatore intero al prossimo fratello. Un fratello è un nodo avente lo stesso nome.

Nodo.java.Vector parsing.Nodo.sons [package]

Vettore di puntatori ai figli del nodo.

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/parsing/Nodo.java

classe parsing.ParseTransaction

Membri pubblici statici

- static void main (String[] args)
- static void CreaFP_Tree ()
- static Hashtable FP_Growth (FP_Tree FP)
- static boolean Incompatible (String key)
- static void SaveKPattern ()

Attributi pubblici statici

- static int numRic = 0
- static final int SUPPORTO = 60
- static int minSup = 0
- static Vector TT = new Vector()
- static Hashtable KP = new Hashtable()
- static Vector path = new Vector()
- static Hashtable MKP = new Hashtable()
- static Hashtable tmpItemsHashtable = new Hashtable()
- static Vector itemsVector = new Vector()
- static Vector codeVector = new Vector()
- static Vector itemsFrequenti = new Vector()
- static int nextItemCode = 1
- static FP_Tree FP3 = new FP_Tree()

Attributi statici con visibilità di package

- static int tot = 0

Membri privati statici

- static void createTable1_1 ()
- static String codify (String l)
- static String generateNextCode ()
- static String modifyCode (String oldCode)
- static int searchString (String str, Vector v)
- static void createtable3 ()
- static void OrdinamentoTabella1_2 ()
- static boolean CodiciUguali (String a, String b)

Attributi privati statici

- static final int MAX_REPLICHE = 30

Descrizione Dettagliata

Implementazione del metodo di parsing delle transazioni generate dal plug-in. Determina l'insieme dei k-predicati frequenti, in base alle relazioni fornite in input (attraverso il file "SpatialTransactions.txt") e al valore percentuale di minimo supporto richiesto.

Autore:

Luigi Scrimitore (scrimito@cli.di.unipi.it)

Documentazione delle funzioni membro

static boolean parsing.ParseTransaction.CodiciUguali (String *a*, String *b*) [static, private]

Determina se due item hanno lo stesso codice, ovvero se sono relativi alla stessa relazione spaziale e differiscono al più per il codice relativo al numero di ripetizioni.

Parametri:

String *a*, *Codice del primo item*
String *b* *Codice del del secondo item*

Definizione alla linea 308 del file ParseTransaction.java.308 {

```
309             // es 44#2 ==> part1 =44; part2 = 2;
310             String part1_a = a.substring(0, a.lastIndexOf('#'));
311             String part2_a = a.substring(a.lastIndexOf('#')+1);
312             int freq_a = Integer.valueOf(part2_a).intValue();
313
314             String part1_b = b.substring(0, b.lastIndexOf('#'));
315             String part2_b = b.substring(b.lastIndexOf('#')+1);
316             int freq_b = Integer.valueOf(part2_b).intValue();
317
318             if ((part1_a.compareTo(part1_b)== 0) && (freq_a >= freq_b)){
319                 return true;
320             }
321             return false;
322         }
```

static String parsing.ParseTransaction.codify (String *l*) [static, private]

Genera un codice per ogni nuova relazione spaziale passata come parametro. Se la relazione è nota fornisce il codice già assegnato.

Parametri:

String *l* *Relazione spaziale da codificare*

Definizione alla linea 169 del file ParseTransaction.java.169 {

```
170             if (tmpItemsHashtable.containsKey(l))
171                 return (String)tmpItemsHashtable.get(l);
172
173             String code = generateNextCode();
174             itemsVector.addElement(l);
175             codeVector.addElement(code);
176             tmpItemsHashtable.put(l, code);
177             return code;
178         }
```

static void parsing.ParseTransaction.CreaFP_Tree () [static]

Crea un oggetto FP_Tree in base alle transazioni note.

```
Definizione alla linea 323 del file ParseTransaction.java.323                    {
324             // creazione dello header dell'albero
325             for (int i=0; i<itemsFrequenti.size(); i++){
326                   Item it = (Item) itemsFrequenti.get(i);
327                   FP3.loadHeader(it.codice, it.freq);
328             }
329
330             for(int i=0; i<TT.size(); i++){
331                   Vector v = (Vector) TT.elementAt(i);
332                   FP3.insertTransaction(v,1);
333             }
334
335             // stampa del Frequent pattern Tree
336             System.out.println("  numero di nodi ... "+ FP3.n.size());
337 //          FP3.print();
338             System.out.println();
339             System.out.println("------------------------------------------------------------");
340
341             minSup = (SUPPORTO*     TT.size())/100;
342             // libero un pò di memoria...
343             TT.clear();
344
345       }
```

static void parsing.ParseTransaction.createTable1_1 () [static, private]

Genera la tabella delle transazioni dal file di testo "*SpatialTransactions.txt*" contenente le relazioni spaziali generate dal plug-in "*SpatialTransactionsGeneratorPlugIn*".

```
Definizione alla linea 82 del file ParseTransaction.java.82                      {
83              Hashtable NOR = new Hashtable();
84              try {
85                    // lettura file delle transazioni
86                    FileReader fr = new FileReader("SpatialTransactions.txt");
87                    BufferedReader br = new BufferedReader(fr);
88
89                    int newTrans=0;
90                    // inizializzazioni
91                    String line = br.readLine();
92                    String line2 = null;
93                    int pos = 0;
94                    TT.addElement(new Vector());
95
96                    // parsing ....
97                    while (line != null) {
```

```
98                      line2 = br.readLine();
99                      if (line.startsWith(" **** ")) {
100                             //inizio nuovo layer
101                             tmpItemsHashtable.clear();
102                             System.out.println("\t\t\t\t Nuovo layer " + line);
103
104                     }
105                     else if (line.startsWith("    (")) {
106                             // è un item della transazione
107                             // calcolo il suo codice;
108                             // cerco se è già presente nella transaz. corrente
109                             String TMPCode = codify(line.trim());
110                             int indexOfCode = searchString(TMPCode, (Vector)((Vector)TT.get(pos)));
111                             if ( indexOfCode > -1){
112                                     // item già inserito; modifica della frequenza;
113                                     String TMP = (String)
((Vector)((Vector)TT.get(pos))).get(indexOfCode);
114                                     ((Vector)((Vector)TT.get(pos))).remove(indexOfCode);
115                                     TMP = modifyCode(TMP);
116                                     ((Vector)((Vector)TT.get(pos))).insertElementAt(TMP, indexOfCode);
117                             }
118                             else// codice nuovo
119                                     ((Vector)((Vector)TT.get(pos))).addElement(TMPCode+"#1");
120                     } else {
121                             // per evitare il caso di righe degeneri vuote
122                             if (line2 != null){
123                             // E' un nuovo oggetto referente.
124                             // evito righe degeneri e oggetti referenti senza relazioni
125                             if (line2.startsWith("    (")) {
126                                     // cerco se ho già incontrato questo oggetto
127                                     if (! NOR.containsKey(line.trim())){
128                                             // nuovo oggetto referente
129                                             // carico i dati del precedente.
130                                             NOR.put(line.trim(), ""+newTrans++);
131                                             TT.addElement(new Vector());
132                                             pos = TT.size()-1;
133
134                                     }
135                                     else {// obj referente già noto
136                                             Integer indexOfName = (Integer.valueOf((String)
(NOR.get(line.trim()))));
137                                             pos = indexOfName.intValue();
138                                     }
139                                     }
140                             }
141                     }
142                     line = line2;
143             }
144     } catch (FileNotFoundException e) {
145             e.printStackTrace();
146     } catch (IOException e) {
147             e.printStackTrace();
148     }
149     System.out.println();
```

```
150             System.out.println(" VISUALIZZAZIONE TABELLA 1.1");
151             System.out.println(" ITEMS INCONTRATI "+itemsVector.size());
152             for (int j=0; j<itemsVector.size(); j++){
153                     System.out.print(" "+itemsVector.elementAt(j));
154                     System.out.println(" "+codeVector.elementAt(j));
155             }
156     }
```

static void parsing.ParseTransaction.createtable3 () [static, private]

Genera la tabella 3 del metodo, necessaria per determinare gli items frequenti e le loro ripetizioni frequenti.

```
Definizione alla linea 206 del file ParseTransaction.java.206                   {
207             final int RIGHE = TT.size();
208             final int COLONNE = itemsVector.size();
209
210             int[][] tab_3 = new int[COLONNE][MAX_REPLICHE];
211
212             for (int i=0; i<RIGHE; i++){
213                     Vector trans = (Vector) TT.get(i);
214                     for (int j=0; j< trans.size(); j++){
215                             String codeItem = (String) trans.get(j);
216                             // part1 = codice dell'item;
217                             // part2 = numero di volte che appare in questa transazione
218                             String part1 = codeItem.substring(0, codeItem.lastIndexOf('#'));
219                             String part2 = codeItem.substring(codeItem.lastIndexOf('#')+1);
220                             int index = Integer.valueOf(part1.trim()).intValue()-1;
221                             int frequenza = Integer.valueOf(part2.trim()).intValue();
222
223                             for (int k=1; k <= frequenza; k++){
224                                     if (k<MAX_REPLICHE)
225                                             tab_3[index][k]++;
226                             }
227                     }
228             }
229             // ANALISI DELLE TABELLE
230             // determinazione degli items frequenti
231             System.out.println();
232             System.out.println("RICERCA");
233             int minSup = (SUPPORTO*RIGHE)/100;
234             System.out.println("MIN_SUP ==>"+minSup);
235             for (int i=0; i<COLONNE; i++){
236                     int k = 1;
237                     String name = (String) itemsVector.get(i);
238                     String code = (String) codeVector.get(i);
239                     while ((tab_3[i][k] >= minSup) && (k < MAX_REPLICHE)){
240                             Item it = new Item(     name +"#"+k,
241                                                     code +"#"+k,
242                                                     tab_3[i][k] );
243                             itemsFrequenti.addElement(it);
```

```
244 //                      it.print();
245                         k++;
246                     }
247                 }
248                 System.out.println("# ITEMS FREQUENTI: .."+itemsFrequenti.size());
249
250                 // rilascio un pò di memoria...
251 //              itemsVector.clear();
252 //              codeVector.clear();
253         }
```

---

static Hashtable parsing.ParseTransaction.FP_Growth (FP_Tree *FP*) [static]

Metodo ricorsivo di visita del FP-Tree e generatore dei k-predicati frequenti presenti nel albero. Per ogni item da analizzare si generano tre diversi alberi:

- *Conditional FP-Tree*
- *Single Path FP-Tree*
- *Multi Path FP-Tree*

Parametri:

FP_Tree *FP* *oggetto FP-Tree.*

```
Definizione alla linea 354 del file ParseTransaction.java.354                                   {
355             // profondità di ricorsione = numRic
356             numRic++;
357             int freq = 0;
358             Hashtable allKPath = new Hashtable();
359             for (int i=FP.h.size()-1; i>=0; i--){
360                 Vector tmpTT = new Vector();
361                 Vector tmpF = new Vector();
362                 Hashtable tmpIF = new Hashtable();
363                 Hashtable KSPath = new Hashtable();
364                 Hashtable KMPath = new Hashtable();
365
366                 String ITEM = ((Head) FP.h.get(i)).NameItem;
367                 int freqItem = ((Head) FP.h.get(i)).freq;
368                 allKPath.put(ITEM, ""+freqItem);
369                 // prendo il 1° nodo puntato nel Header
370                 int cnt = ((Head) FP.h.get(i)).firstPosition;
371
372                 if (numRic == 1){
373                     System.out.println(" ITEM : "+(i+1)+"-esimo :"+ITEM+" freq :"+freqItem+" fst
pos ="+cnt);
374                     for (Enumeration e = allKPath.keys(); e.hasMoreElements(); ){
375                         String key = (String) e.nextElement();
376                         KP.put(key, (String) allKPath.get(key));
377                     }
378                     allKPath.clear();
379                 }
380                 while (cnt > -1){
```

```
381                         // risalgo l'albero creando un vettore
382                         // in cui registro tutti gli items che incontro
383                         // la frequenza di questo cammino è uguale a quello del nodo posto più in basso
384                         // ovvero quello di partenza
385                         Vector tmpTrans = new Vector();
386
387                         int pos = ((Nodo) FP.n.get(cnt)).dad;
388                         freq = ((Nodo) FP.n.get(cnt)).freq;
389
390                         // finchè non giungo al nodo radice
391                         while (pos >0){
392                               int supp = freq;
393                               Nodo NodoCorrente = (Nodo) FP.n.get(pos);
394                               String Name = (String) NodoCorrente.Name;
395                               if (tmpIF.containsKey(Name)){
396                                     supp += (Integer.valueOf((String) (tmpIF.get(Name)))).intValue();
397                                     tmpIF.remove(Name);
398                               }
399                               tmpIF.put(Name, ""+supp);
400                               tmpTrans.addElement(Name);
401                               pos = NodoCorrente.dad;
402                         }
403                         tmpF.addElement(""+freq);
404                         tmpTT.addElement(tmpTrans);
405                         cnt = ((Nodo) FP.n.get(cnt)).nextBrother;
406                   }
407
408                   // creazione "Conditional FP_Tree" relativo al Item ((Head) FP3.h.get(i)).NameItem
409                   FP_Tree newFP3 = new FP_Tree();
410                         for (Enumeration e = tmpIF.keys(); e.hasMoreElements(); ){
411                               Object key = e.nextElement();
412                               int supp = (Integer.valueOf((String) (tmpIF.get(key)))).intValue();
413                               if (supp < minSup)
414                                     tmpIF.remove(key);
415                               else newFP3.loadHeader((String) key, supp);
416                         }
417                         for (int k=0; k< tmpTT.size(); k++){
418                               Vector t = (Vector) tmpTT.get(k);
419                               Vector tt = new Vector();
420                               int dim = t.size();
421                               int supp = (Integer.valueOf((String) tmpF.get(k))).intValue();
422                               for (int j=dim-1; j>=0; j--){
423                                     // eliminazione items infrequenti
424                                     String key = (String) t.get(j);
425                                     if (tmpIF.containsKey(key)){
426                                           tt.addElement(key);
427                                     }
428                               }
429                               newFP3.insertTransaction(tt, supp);
430                         }
431
432 //                  System.out.println();
433 //                  System.out.println("CONDITIONAL fp3"+"# NODI :"+newFP3.n.size()+"\t# item
freq : "+newFP3.h.size());
```

```
434 //                  newFP3.print();
435
436                         // SINGLE PATH FP_TREE
437                     Vector SP_FP3 = new Vector();
438                     Vector SP_freq = new Vector();
439                     int dad = 0;
440                     //      se esiste Single Path FP3
441                     //  calcolo del SinglePathFP3
442                     while (((Nodo)newFP3.n.get(dad)).sons.size() == 1 ) {
443                         int ind = (Integer.valueOf((String)
((Nodo)newFP3.n.get(dad)).sons.get(0))).intValue();
444                         Nodo onlySon = (Nodo) newFP3.n.get(ind);
445                         SP_FP3.addElement(onlySon.Name);
446                         SP_freq.addElement(""+onlySon.freq);
447                         dad++;
448                     }
449                     // calcolo l'insieme dei suoi K-predicati (Single path set P)
450                     for (int k=0; k<SP_FP3.size(); k++){
451                         int dim = KSPath.size();
452                         String TMPs = (String) SP_FP3.get(k);
453                         for (Enumeration e = KSPath.keys(); e.hasMoreElements(); ){
454                             String key = (String) e.nextElement();
455                             String newKey = TMPs+","+key;
456                             KSPath.put(newKey, SP_freq.get(k));
457                             allKPath.put(ITEM+","+newKey, ""+freqItem);
458                         }
459                         allKPath.put(ITEM+","+TMPs, ""+freqItem);
460                     }
461                     for (int k=0; k<SP_FP3.size(); k++){
462                         String TMPs = (String) SP_FP3.get(k);
463                         KSPath.put(TMPs, SP_freq.get(k));
464                     }
465
466                     // MULTI PATH FP_TREE
467                     FP_Tree MP_FP3 = new FP_Tree();
468                     if ((newFP3.n.size()-dad-1) >0){
469                         MP_FP3.CloneFP3(newFP3, dad);
470 //                      MP_FP3.print();
471
472 //                      chiamata ricorsiva di FP_Growth() su MP_FP3
473                         KMPath = (Hashtable) FP_Growth(MP_FP3);
474                         numRic--;
475                         //########## RITORNO DALLA RICORSIONE ############
476
477                         for (Enumeration e = KMPath.keys(); e.hasMoreElements(); ){
478                             String key = (String) e.nextElement();
479                             int supp = (Integer.valueOf((String) KMPath.get(key))).intValue();
480 //                          System.out.println("KEY "+key+" SUPP ="+supp);
481                             if ( supp > minSup){
482                                 String newKey = ITEM +","+ key;
483                                 allKPath.put(newKey, ""+freqItem);
484                                 for (Enumeration ee = KSPath.keys(); ee.hasMoreElements(); ){
485                                     String key2 = (String) ee.nextElement();
486                                     String newKey2 = newKey+","+key2;
```

```
487                                     allKPath.put(newKey2, ""+supp);
488                                 }
489                             }
490                         }
491
492                         // cancellazione duplicati
493                         for (Enumeration e = allKPath.keys(); e.hasMoreElements(); ){
494                             String key = (String) e.nextElement();
495                             int supp = (Integer.valueOf((String) allKPath.get(key))).intValue();
496                             if (Incompatible(key))
497                                 allKPath.remove(key);
498                         }
499                     }
500
501             }
502             if (numRic == 1)
503                 return KP;
504                 else return allKPath;
505         }
```

---

static String parsing.ParseTransaction.generateNextCode () [static, private]

Generatore del prossimo codice di un item. E' un numero progressivo.

```
Definizione alla linea 180 del file ParseTransaction.java.180                    {
181             String TMP = ""+ nextItemCode;
182             nextItemCode++;
183             return TMP;
184         }
```

---

static boolean parsing.ParseTransaction.Incompatible (String *key*) [static]

Verifica se un K-Pattern ha items incompatibili. Due items sono incompatibili se sono relativi alla stessa relazione spaziale, ovvero se hanno il primo codice uguale.

Parametri:

String *key* *K-Pattern da analizzare.*

```
Definizione alla linea 507 del file ParseTransaction.java.507                    {
508             int min = 0;
509             int max = 0;
510             while (key.contains(",")){
511                 max = key.indexOf("#");
512                 String str = ","+key.substring(0, max+1);
513                 int pos = key.indexOf(",");
514                 if (pos > max){
515                     String Path = key.substring(pos);
516                     if (Path.contains(str)){
517                         // ******************** PATTERN INCOMPATIBILE
```

```
518                             return true;
519                         }
520                         key = key.substring(pos+1);
521                     } else return false;
522                 }
523                 return false;
524
525         }
```

---

static void parsing.ParseTransaction.main (String[] *args*) [static]

Main della classe. Richiama in sequenza i metodi che implementano le fasi del processo di parsing.

- CreateTable_1
- CreateTable3
- OrdinamentoTabella1_2
- CreaFP_Tree
- FP_Growth

Parametri:

String[] *args* *vettore degli argomenti.*

```
Definizione alla linea 50 del file ParseTransaction.java.50                                 {
51              System.out.println("Creazione Tabella 1.1");
52              createTable1_1();
53              System.out.println("FINE 1");
54              System.out.println();
55              System.out.println("-------------------------------------------------------------");
56              System.out.println("Creazione Tabelle 2 e 3");
57              createtable3();
58
59              System.out.println("-------------------------------------------------------------");
60              System.out.println("Ordinamento items frequenti");
61              OrdinamentoTabella1_2();
62
63              System.out.println();
64              System.out.println("-------------------------------------------------------------");
65              System.out.println("CREAZIONE FP-Tree");
66              CreaFP_Tree();
67
68              System.out.println("Ricerca dei k-pattern frequenti in corso...");
69              System.out.println("minimo supporto ="+minSup);
70              KP = (Hashtable) FP_Growth(FP3);
71
72              System.out.println("# K-Patterns frequenti : "+KP.size());
73              SaveKPattern();
74
75              System.out.println("*****************************************************");
76              System.out.println("**************  PARSING ULTIMATO!  ****************");
77              System.out.println("*****************************************************");
```

```
78
79      }
```

---

static String parsing.ParseTransaction.modifyCode (String *oldCode*) [static, private]

Modifica il codice di un item, ovvero incrementa di uno il secondo codice, quello relativo al numero di ripetizioni del item.

Parametri:

String *oldCode* *codice item da incrementare.*

```
Definizione alla linea 186 del file ParseTransaction.java.186                    {
187             int TMP = oldCode.lastIndexOf("#");
188             int numRipetizioni = Integer.valueOf( oldCode.substring(TMP+1) ).intValue();
189             numRipetizioni++;
190             String newCode = oldCode.substring(0, TMP+1) + numRipetizioni;
191             return newCode;
192     }
```

---

static void parsing.ParseTransaction.OrdinamentoTabella1_2 () [static, private]

Ordinamento del vettore degli items frequenti e aggiornamento della tabella delle transazioni "*TT*". L'aggiornamento del dataset si effettua sostituendo ogni transazione con l'insieme ordinato (ordinamento decrescente) degli items frequenti che contiene.

```
Definizione alla linea 254 del file ParseTransaction.java.254                    {
255             // ordinamento del vettore degli items frequenti
256             // Bubble Sort;
257             boolean ordina = true;
258             while (ordina){
259                     ordina = false;
260                     for (int i=0; i<itemsFrequenti.size()-1; i++){
261                             Item it1 = (Item) itemsFrequenti.get(i);
262                             Item it2 = (Item) itemsFrequenti.get(i+1);
263                             if (it1.freq < it2.freq){
264                                     ordina = true;
265                                     itemsFrequenti.remove(i);
266                                     itemsFrequenti.insertElementAt( it1, i+1);
267                             }
268                     }
269             }
270
271             for (int i=0; i<itemsFrequenti.size(); i++){
272                     ((Item)itemsFrequenti.get(i)).print();
273             }
```

```
274             System.out.println();
275             System.out.println("------------------------------------------------------------");
276             System.out.println("TABELLA ORDINATA :");
277
278             boolean[] BoolTempVect = new boolean[itemsFrequenti.size()];
279             for (int z=0; z<BoolTempVect.length; z++){
280                 BoolTempVect[z] = false;
281             }
282
283             System.out.println("TT.size() = "+TT.size());
284             for (int i=0; i<TT.size(); i++){
285                 Vector t = (Vector) TT.get(i);
286                 for (int j=0; j<t.size(); j++){
287                     // ordina TRANSAZIONE con le ripetizioni
288                     int k=0;
289                     while (k < itemsFrequenti.size()){
290                         String a = (String) t.get(j);
291                         String b = ((Item)itemsFrequenti.get(k)).codice;
292                         if (CodiciUguali(a,b))
293                             BoolTempVect[k] = true;
294                         k++;
295                     }
296                 }
297                 t.clear();
298                 for (int z=0; z<BoolTempVect.length; z++){
299                     if (BoolTempVect[z]){
300                         t.addElement(((Item)itemsFrequenti.get(z)).codice);
301                         BoolTempVect[z] = false;
302                     }
303                 }
304             }
305
306     }
```

static void parsing.ParseTransaction.SaveKPattern () [static]

Salvataggio dell'insieme di K-Pattern frequenti generati da *FP-Growth()* in un file di testo (*K_Patterns.txt*).

```
Definizione alla linea 528 del file ParseTransaction.java.528                    {
529             Hashtable htif = new Hashtable();
530             for (int i =0; i<itemsFrequenti.size();i++){
531                 Item it = ((Item) itemsFrequenti.get(i));
532                 String key = it.codice;
533                 htif.put(key, it.Nome);
534             }
535
536             PrintWriter out = null;
537             try {
538                 int tot=0;
539                 out = new PrintWriter(new FileWriter("K_Patterns.txt"));
```

```
540                 for (Enumeration e = KP.keys(); e.hasMoreElements(); ){
541                     String key = (String) e.nextElement();
542                     String Nome = "";
543                         tot++;
544                         while (key.contains(",")){
545                             int max = key.indexOf(",");
546                             String key2 = key.substring(0, key.indexOf(","));
547                             key = key.substring(key.indexOf(",")+1);
548                             Nome = (String) htif.get(key2);
549                             out.print("\t"+Nome);
550                             for (int j=Nome.length(); j<55; j++){
551                                 out.print(" ");
552                             }
553                         }
554                         Nome = (String) htif.get(key);
555                         out.print("\t"+Nome);
556                         out.println("\t freq = "+KP.get(key));
557                 }
558
559                 for (int j=0; j<itemsVector.size(); j++){
560                     out.println("**********************  HO FINITO!  **************"+tot);
561                 }
562             } catch (IOException ex) {
563                 ex.printStackTrace();
564             }
565             System.out.println("# ITEMS SENZA RIPETIZIONI FREQUENTI : "+htif.size());
566         }
```

static int parsing.ParseTransaction.searchString (String *str*, Vector *v*) [static, private]

Ricerca in una transazione "*v*" la posizione del item avente codice "*str*". *Se non è presente ritorna* "-1".

Parametri:

String *str*,	*codice item da cercare*
Vector *v*	*vettore di items. Rappresenta una transazione della tabella TT.*

```
Definizione alla linea 194 del file ParseTransaction.java.194                                    {
195             for (int i=0; i<v.size(); i++){
196                 String TMP = (String) v.get(i);
197                 if (TMP.indexOf("#")> -1 )
198                     TMP = TMP.substring(0, TMP.lastIndexOf("#"));
199                 if (str.equals(TMP))
200                     // stringa già inserita; si tratta di una replica
201                     return i;
202             }
203             return -1;
204         }
```

Documentazione dei dati membri

Vector parsing.ParseTransaction.codeVector = new Vector() [static]

Vettore in cui sono memorizzati i codici degli items.

ParseTransaction.java.FP_Tree parsing.ParseTransaction.FP3 = new FP_Tree() [static]

Oggetto FP_Tree.

ParseTransaction.java.Vector parsing.ParseTransaction.itemsFrequenti = new Vector() [static]

Vettore degli items frequenti.

ParseTransaction.java.Vector parsing.ParseTransaction.itemsVector = new Vector() [static]

Vettore degli items trovati.

ParseTransaction.java.Hashtable parsing.ParseTransaction.KP = new Hashtable() [static]

Hashtable di K-Pattern. La chiave è il codice dell'pattern l'oggetto la sua frequenza.

ParseTransaction.java.final int parsing.ParseTransaction.MAX_REPLICHE = 30 [static, private]

Costante che determina il numero massimo di repliche di un item consentite.

ParseTransaction.java.int parsing.ParseTransaction.minSup = 0 [static]

Minimo supporto consentito.

ParseTransaction.java.Hashtable parsing.ParseTransaction.MKP = new Hashtable() [static]

ParseTransaction.java.int parsing.ParseTransaction.nextItemCode = 1 [static]

Prossimo codice disponibile da assegnare agli items.

ParseTransaction.java.int parsing.ParseTransaction.numRic = 0 [static]

Livello di ricorsione di FP_Growth()

ParseTransaction.java.Vector parsing.ParseTransaction.path = new Vector() [static]

ParseTransaction.java.final int parsing.ParseTransaction.SUPPORTO = 60 [static]

Vincolo percentuale di minimo supporto.

ParseTransaction.java.Hashtable parsing.ParseTransaction.tmpItemsHashtable = new Hashtable() [static]

Hashtable degli items incontrati per il layer corrente.

ParseTransaction.java.int parsing.ParseTransaction.tot = 0 [static, package]

ParseTransaction.java.Vector parsing.ParseTransaction.TT = new Vector() [static]

Transaction Table

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/parsing/ParseTransaction.java

classe plugins.SpatialTransactionsGeneratorExtension

Membri pubblici

- void configure (PlugInContext context) throws Exception

Descrizione Dettagliata

Estensione del plug-in SpatialTransactionsGenerator.

Autore:

Salvatore Rinzivillo (rinziv@di.unipi.it)

Documentazione delle funzioni membro

void plugins.SpatialTransactionsGeneratorExtension.configure (PlugInContext *context*) throws Exception

Parametri:

context *plug-in context*

Definizione alla linea 17 del file SpatialTransactionsGeneratorExtension.java.17

```
{
18        new SpatialTransactionsGeneratorPlugIn().initialize(context);
19    }
```

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/plugins/SpatialTransactionsGeneratorExtension.java

classe plugins.SpatialTransactionsGeneratorPlugIn

Membri pubblici

- boolean execute (PlugInContext context)
- void run (TaskMonitor monitor, PlugInContext context) throws Exception
- void initialize (PlugInContext context) throws Exception

Membri protetti

- void generateSpatialTransaction (Layer training, Hashtable attributeMapping, Vector layers, PrintWriter out)

Attributi privati

- MultiInputDialog dialog

Descrizione Dettagliata

Plug-in di JUMP per la generazione di transazioni spaziali.

Autore:

Salvatore Rinzivillo (rinziv@di.unipi.it)

Documentazione delle funzioni membro

boolean plugins.SpatialTransactionsGeneratorPlugIn.execute (PlugInContext *context*)

metodo di comando di esecuzione della finestra di dialogo.

Parametri:

context *plug-in context*

```
Definizione alla linea 40 del file SpatialTransactionsGeneratorPlugIn.java.40
{
41              Layer[] selectedLayers = context.getSelectedLayers();
42
43              dialog = new MultiInputDialog(context.getWorkbenchFrame(), getName(),
44                              true);
45              dialog
46                              .setSideBarDescription("Select the layers to use for generating the spatial
transactions");
47              // Add a checkbox for selecting the reference layer
48              dialog.addLayerComboBox("Reference Layer",
49                              context.getCandidateLayer(0),
50                              "This Layer will be used as reference layer.", context
51                                              .getLayerManager());
52
53              // Popupate the dialog windows with the info of layers and their
54              // attributes
55              for (int i = 0; i < selectedLayers.length; i++) {
56                      Layer currLayer = selectedLayers[i];
57                      //out.addHeader(2,"Layer: " + currLayer.getName());
```

```
58                  FeatureCollection fc = currLayer.getFeatureCollectionWrapper();
59                  FeatureSchema fs = fc.getFeatureSchema();
60                  Vector attributes = new Vector();
61                  for (int j = 0; j < fs.getAttributeCount(); j++) {
62                        String attrName = fs.getAttributeName(j);
63                        //out.addField("Attribute " + j +": ",attrName);
64                        attributes.add(attrName);
65                  }
66                  dialog.addComboBox("Attribute " + currLayer.getName(), currLayer
67                              .getName().toUpperCase(), attributes,
68                              "Choose one of the attributes as categorical attribute");
69            }
70
71            dialog.setVisible(true);
72            System.out.print("Ho FINITO EXECUTE");
73            if (!dialog.wasOKPressed()) {
74                  return false;
75            }
76
77
78            return true;
79      }
```

void plugins.SpatialTransactionsGeneratorPlugIn.generateSpatialTransaction (Layer *training*, Hashtable *attributeMapping*, Vector *layers*, PrintWriter *out*) [protected]

Generatore delle relazioni spaziali.

Parametri:

training	*layer referente*
attributeMapping	*Hashtable con i nomi degli attributi scelti.*
layers	*vettore dei layer referenziati*
out	*file di output*

```
Definizione alla linea 115 del file SpatialTransactionsGeneratorPlugIn.java.116
{
117             System.out.println("LAYER"+training.getName());
118             // TODO: da spostare in una classe indipendente
119             String refAttr = (String) attributeMapping.get(training.getName());
120             for (Iterator i = layers.iterator(); i.hasNext();) {
121                   Layer l = (Layer) i.next();
122                   String attr = (String) attributeMapping.get(l.getName());
123
124                   System.out.println(" **** Layer " + l.getName() + " **** ");
125                   out.println(" **** Layer " + l.getName() + " **** ");
126                   int tot = 1;
127
128                   IndexedFeatureCollection ifc = new IndexedFeatureCollection(
129                               l.getFeatureCollectionWrapper());
```

```
130                 for (Iterator j = training.getFeatureCollectionWrapper().iterator(); j
131                         .hasNext();) {
132                     Feature f = (Feature) j.next();
133                     Geometry rg = f.getGeometry().buffer(0.03d);
134
135                     out.println(f.getAttribute(refAttr).toString());
136                     System.out.println(tot++);
137
138                     List lst = ifc.query(rg.getEnvelopeInternal());
139                     for (Iterator k = lst.iterator(); k.hasNext();) {
140                         Feature f1 = (Feature) k.next();
141                         Geometry g1 = f1.getGeometry();
142
143                         if (g1.getEnvelopeInternal().intersects(
144                                 rg.getEnvelopeInternal())) {
145                             IntersectionMatrix im = rg.relate(g1);
146                             String relation = null;
147                             if (im.isContains())
148                                 relation = "contains";
149                             else if (im.isCrosses(rg.getDimension(), g1
150                                     .getDimension()))
151                                 relation = "crosses";
152                             //                                  else if (im.isDisjoint())
153                             //                                      relation = "disjoint";
154                             else if (im.isIntersects())
155                                 relation = "intersects";
156                             else if (im.isOverlaps(rg.getDimension(), g1
157                                     .getDimension()))
158                                 relation = "overlaps";
159                             else if (im.isTouches(rg.getDimension(), g1
160                                     .getDimension()))
161                                 relation = "touches";
162                             else if (im.isWithin())
163                                 relation = "within";
164
165                             if (relation != null) {
166                                 out.println("    (" + relation + ", "+ l.getName() +":\""
167                                         + f1.getAttribute(attr).toString().trim()
168                                         + "\")");
169                             }
170                         }
171                     }
172                 }
173             }
174         }
```

void plugins.SpatialTransactionsGeneratorPlugIn.initialize (PlugInContext *context*) throws Exception

Metodo di inizializzazione della finestra di dialogo.

Parametri:

context *plug-in context.*

```
Definizione alla linea 176 del file SpatialTransactionsGeneratorPlugIn.java.176
{
177             context.getFeatureInstaller().addMainMenuItem(
178                     this,
179                     new String[] { "Tools", "LUX" },
180                     getName(),
181                     false,
182                     null,
183                     new EnableCheckFactory(context.getWorkbenchContext())
184                             .createAtLeastNLayersMustBeSelectedCheck(2));
185     }
```

---

void plugins.SpatialTransactionsGeneratorPlugIn.run (TaskMonitor *monitor*, PlugInContext *context*) throws Exception

Metodo di esecuzione della finestra di dialogo.

Parametri:

Monitor, *gestore JUMP dei processi interni.*

context *plug-in context.*

```
Definizione alla linea 81 del file SpatialTransactionsGeneratorPlugIn.java.82                {
83
84              Layer[] selectedLayers = context.getSelectedLayers();
85              monitor.allowCancellationRequests();
86              Layer training = dialog.getLayer("Reference Layer");
87
88              // Retrieve the list of attributes for the layers
89              Hashtable attributeMapping = new Hashtable();
90              Vector layers = new Vector();
91              for (int i = 0; i < selectedLayers.length; i++) {
92                    Layer currLayer = selectedLayers[i];
93                    attributeMapping.put(currLayer.getName(), dialog
94                                .getText("Attribute " + currLayer.getName()));
95                    if (!(currLayer.getName().equals(training.getName()))) {
96                          layers.add(currLayer);
97                    }
98              }
99              PrintWriter out = null;
100              try {
101                    out = new PrintWriter(new FileWriter("SpatialTransactions.txt"));
102              } catch (IOException e) {
103                    e.printStackTrace();
104              }
105              generateSpatialTransaction(training, attributeMapping, layers, out);
```

```
106             System.out.print("Ho FINITO TUTTO IL RUN");
107
108     }
```

Documentazione dei dati membri

MultiInputDialog plugins.SpatialTransactionsGeneratorPlugIn.dialog [private]

Finestra di dialogo.

La documentazione per questa classe è stata generata a partire dal seguente file:

- KDonGIS/plugins/SpatialTransactionsGeneratorPlugIn.java

KDDonGIS Documentazione dei file

KDonGIS/parsing/FP_Tree.javaNamespace

- namespace parsing

Composti

- class parsing.FP_Tree

KDonGIS/parsing/Head.javaNamespace

- namespace parsing

Composti

- class parsing.Head

KDonGIS/parsing/Item.javaNamespace

- namespace parsing

Composti

- class parsing.Item

KDonGIS/parsing/Nodo.javaNamespace

- namespace parsing

Composti

- class parsing.Nodo

KDonGIS/parsing/ParseTransaction.javaNamespace

- namespace parsing

Composti

- class parsing.ParseTransaction

KDonGIS/plugins/SpatialTransactionsGeneratorExtension.javaNamespace

- namespace plugins

Composti

- class plugins.SpatialTransactionsGeneratorExtension

KDonGIS/plugins/SpatialTransactionsGeneratorPlugIn.javaNamespace

- namespace plugins

Composti

- class plugins.SpatialTransactionsGeneratorPlugIn

Indice

Bibliografia

[AAS01] **Integration of GIS with Data Mining**
G. L. Andrienko, N. V. Andrienko, A. A. Savinov.
In ETK-99. Exchange of Technology and Know-How Statistical Office of the European Communities, Luxembourg, 2001, pp.221-225. ISBN 92-828-7883-X

[AAS94] **Fast algorithm for mining association rules**
R.Agrawall, R.Srikant.
In Proc. 20th Int. Conf. Very Large Data Bases, VLDB, pages 487-- 499. Morgan Kaufmann, 12--15 1994.

[BK90] **The R*-tree: An Efficient and Robust Access Method for Points and Rectangles**
N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger.
In the Proceedings of ACM-SIGMOD Int. Conference on Management of Data, pp. 322-331, Atlantic City, USA, 1990.

[DOXY] **http://www.doxygen.org/index.html**
Copyright © 1997-2005 by Dimitri van Heesch.

[EF91] **Point-set topological spatial relations**
Max J. Egenhofer and Robert D. Franzosa.
International Journal of Geographical Information Systems, 5(2):161-176, 1991.

[EFKS00] **Spatial Data Mining: Datbase primitives, algorithms and Efficient DBMS Support.**
M. Ester, A. Frommelt, H. Kriegel and J. Sander.
Data Mining and Knowledge Discovery, 4(2/3):193-216, 2000.

[EH91] **Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases**

Max J. Egenhofer and J. Herring.
Technical Report, Department of Surveying Engineering, University of Maine, 1991.

[EKS97] **Spatial Data Mining: A Database Approach**
M. Ester, H. P. Kriegel and J. Sander.
In Proc. of the Fifth Int. Symposium on Large Spatial Databases (SSD '97), Berlin, Germany, Lecture Notes in Computer Science,Springer, 1997.

[ES93] **Topological relations between regions in IR2 and ZZ2.**
Max J. Egenhofer and J. Sharma.
In: Abel, D. and Ooi, B. C. (eds.), Advances in Spatial Databases (SSD'93), (Lecture Notes in Computer Science, No. 692), Springer-Verlag. 316-226, 1993.

[F91] **Fundamentals of Geographical Information Systems: Hardware, Software and Data**
Bill Fritsch.
In Wichmann Publishing, Heidelberg, Germany, 1991.

[FPM91] **Knowledge Discovery in Databases: An Overview**
W.J. Frawley, G. Piatetsky-Shapiro and J. Matheus.
In Knowledge Discovery in Databases, AAAI Press, Menlo Park, 1991, pp. 1-27.

[GG98] **Multidimensional Access Methods**
V. Gaede and O. Gunther.
In ACM Computing Surveys, Vol. 30, No. 2, pp. 170 – 231, June 1998.

[G84] **R-Trees. A Dynamic Index Structure for Spatial Searching**
Antonin Guttman.
In Proc ACM SIGMOD Int Conf on Management of Data, 47-57, 1984.

[HP04] **Mining Frequent Patterns without Candidate generation: A Frequent-Pattern Tree Approach**
JiaWei Han, Jian Pei, Yiwen Yin and Runying Mao.

In Data Mining and Knowledge Discovery: An International Journal, 8(1):53-87, 2004.

[HK98] **Mining Association Rules in Large Databases**
J. Han and M. Kamber.
In Data Mining: Concepts and Techniques, San Francisco: Morgan Kaufman, 2003.

[HKS98] **GeoMiner: A system prototype for spatial data mining**
J. Han K. Koperski N. Stefanovic.
In Proceedings ACM SIGMOD Conference, pp.553-556, Tucson, Arizona, May 1997.

[JUMP] **http://www.jump-project.org**
VividSolution, "JUMP Workbench," 2003.

[KH95] **Discovery of Spatial Association Rules in Geographic Information Databases**
Krzysztof Koperski and Jiawei Han.
In Advances in Spatial Databases, Proceedings of 4th Symposium, SSD' 95 Aug. 6-9, Portland, Maine. Springer-Verlag, berlin, 1995, pp47-66.

[KAH95] **Spatial Data Mining: Progress and Challenges**
K. Koperski, J. Adhikary and J. Han.
InProceedings Workshop on Research Issues on Data Mining and Knowledge Discovery, Montreal,Canada. 1996.

[K01] **Data structures for spatial data mining**
Petr Kuba.
Department of Computer Science, Faculty of Informatics Masaryk University Brno, Czech Republic, September 4, 2001.

[LNS04] **Developing an Open Source GIS Desktop Application for District Health Information System (DHIS)**
Juma Lungo, Kristian Nordal, Henrik Solgaard.
In Department of informatics, University of Oslo, 24th November 2004.

[MEL01] **Mining Spatial Association Rules in Census Data**

Donato Malerba, Floriana Esposito and Francesca A. Lisi.
In Proc. Of the Joint Conf. On "New Techniques and Technologies for Statistics" and "Exchange of technologie and Know-how", 2001.

[NH 94] **Efficient and Effective Clustering Methods for Spatial Data Mining**
R.T. Ng, J. Han.
In Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile, 1994, pp. 144-155.

[PTSE95] **Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees**
Dimitris Papadias, Yannis Theodoridis, Timos Sellis and Max J. Egenhofer.
In *Proceedings ACM SIGMOD Conference, pp.92-103, San Jose, CA, 1995.*

[R] **A Constraint-Based Spatial Knowledge Discovery Framework**
Salvatore Rinzivillo.
In proposta di tesi.

[RT04] **Classification in Geographic Information Systems**
Salvatore Rinzivillo, Franco Turini.
In 8th European Conference on Principles and Practice of KnowledgeDiscovery in Databases (PKDD 2004), pages 374–385, 2004.

[S84] **The Quadtree and Related Hierarchical Data Structures**
Hanan Samet.
In ACM Computing Surveys, 187-260, June 1984.

[SR87] **The R+-Tree: a Dynamic Index for Multi-Dimensional Objects**
Timos Sellis, Nick Roussopoulos and Christos Faloutsos.
In Proceeding of the 13th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brighton, UK, September 1987.